CloudGraph

P. Baillehache

October 30, 2018

Contents

Introduction

CloudGraph is a C library of functions generating a 2D graphical representation of a graph based on the relations between its nodes. Two types of relation are considered: edges of the graph, and categories of nodes in the graph.

It also provides a front end which reads the graph definition from a text file or generate a random one, produces a TGA picture representing the network, and/or prints the nodes' 2D coordinates.

The representation of the graph has 2 modes: circular and linear. The representation of the links has 2 modes: straight line and curved line. Categories are represented by different color, and links between two categories have shading colors. Nodes and categories are also identified by labels which can be displayed.

1 Input

The description of the CloudGraph is given as input to the front end as a text file. The format of this text file is as follow:

```
<number of category>
for each category: <id> <red> <green> <blue> <label>
<number of node>
for each node: <id> <id of the category> <label>
<number of link>
for each link
<id first node> <id second node>
```

<id> is an integer between 0 and the number of category/node minus one. <red/green/blue> are integers between 0 and 255. <label> is the displayed label, it cannot be empty.

2 Interface

```
// ============ CLOUDGRAPH.H ================

#ifndef CLOUDGRAPH_H
#define CLOUDGRAPH_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "gset.h"
#include "pbmath.h"
#include "bcurve.h"
#include "tgapaint.h"

// ================= Define ==================

#define CG_NBMAXFAMILY 100

// ================= Data structures ==================

// Node of the cloud
typedef struct CloudGraphNode {
  // ID of the node
  int _id;
  // ID of the family of this node
  int _family;
  // Label of this node
  char *_label;
  // Position
  VecFloat *_pos;
  // Bounding box of the node
  Shapoid *_boundingBox;
```

```
  // Bounding box of the label
  Shapoid *_boundingBoxLbl;
  // Vector indicating the right direction from this node
  VecFloat *_right;
  // Angle with the abciss
  float _theta;
} CloudGraphNode;

// Family of the node
typedef struct CloudGraphFamily {
  // ID of the family
  int _id;
  // Color of the family
  unsigned char _rgba[4];
  // Label of this family
  char *_label;
  // Bounding box of the label
  Shapoid *_boundingBox;
  // Position of the label
  VecFloat *_pos;
  // Vector indicating the right direction of the label
  VecFloat *_right;
} CloudGraphFamily;

// Link of the CloudGraph
typedef struct CloudGraphLink {
  // ID of the nodes
  int _nodes[2];
  // BCurve to trace this link
  BCurve *_curve;
  // Bounding box of the link
  Shapoid *_boundingBox;
  // ID of families (for color selection);
  int _families[2];
} CloudGraphLink;

// CloudGraph
typedef struct CloudGraph {
  // SpringSys representing the CloudGraph
  GSet *_nodes;
  // List of families
  GSet *_families;
  // List of links
  GSet *_links;
  // Font to write the labels
  TGAFont *_font;
  // Bounding bos of the cloud
  Shapoid *_boundingBox;
} CloudGraph;

// Modes of CloudGraph representation
typedef enum CloudGraphMode {
  // Default, Nodes are placed along a line
  CloudGraphModeLine,
  // Nodes are placed on the circumference of a circle
  CloudGraphModeCircle
} CloudGraphMode;

// Modes of node's label representation
typedef enum CloudGraphOptNodeLabel {
  // Default, no label
  CloudGraphOptNodeLabelNone,
```

```
  // Label of all nodes
  CloudGraphOptNodeLabelAll
} CloudGraphOptNodeLabel;

// Modes of family's label representation
typedef enum CloudGraphOptFamilyLabel {
  // Default, no label
  CloudGraphOptFamilyLabelNone,
  // Label at the center of the family
  CloudGraphOptFamilyLabelAll
} CloudGraphOptFamilyLabel;

// Graphical options while exporting to TGA
typedef struct CloudGraphOpt {
  // Mode of CloudGraph representation
  CloudGraphMode _mode;
  // Flag to memorize if the links should be represented as curves
  // In linear mode they are always curved
  bool _curvedLink;
  // Curvature in ]0.0, inf[
  float _curvature;
  // Mode for nodes' label
  CloudGraphOptNodeLabel _nodeLabelMode;
  // Mode for families' label
  CloudGraphOptFamilyLabel _familyLabelMode;
  // Font size for nodes
  float _fontSizeNode;
  // Font size for families
  float _fontSizeFamily;
} CloudGraphOpt;

// ================ Functions declaration ====================

// Create a new CloudGraph
// Return NULL if we couldn't create the CloudGraph
CloudGraph* CloudGraphCreate(void);

// Free the memory used by a CloudGraph
// Do nothing if arguments are invalid
void CloudGraphFree(CloudGraph **cloud);

// Free the memory used by a CloudGraphNode
// Do nothing if arguments are invalid
void CloudGraphNodeFree(CloudGraphNode** node);

// Free the memory used by a CloudGraphFamily
// Do nothing if arguments are invalid
void CloudGraphFamilyFree(CloudGraphFamily** family);

// Free the memory used by a CloudGraphLink
// Do nothing if arguments are invalid
void CloudGraphLinkFree(CloudGraphLink** link);

// Set the represention mode to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetMode(CloudGraphOpt *opt, CloudGraphMode mode);

// Create a random CloudGraph having between 'nbNodeMin' and 'nbNodeMax'
// nodes, and between 'nbFamilyMin' and 'nbFamilyMax' families, and
// 'density' (in [0,1]) probability of connection between each pair of
// nodes
// If 'cloud' is not NULL it is first freed
```

```
// The random generator must be initialized before calling this function
// Return true on success, false else (invalid arguments or malloc failed)
bool CloudGraphCreateRnd(CloudGraph **cloud, int nbNodeMin,
  int nbNodeMax, int nbFamilyMin, int nbFamilyMax, float density);

// Create a CloudGraphFamily with default values:
// _id = 0
// _rgba = {0, 0, 0, 255}
// _label = NULL;
// Return NULL if couldn't create the family
CloudGraphFamily* CloudGraphCreateFamily(void);

// Add a copy of the family 'f' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddFamily(CloudGraph *cloud, CloudGraphFamily *f);

// Create a CloudGraphNode with default values:
// _id = 0
// _family = 0
// _label = NULL
// Return NULL if couldn't create the family
CloudGraphNode* CloudGraphCreateNode(void);

// Add a copy of the node 'n' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddNode(CloudGraph *cloud, CloudGraphNode *n);

// Create a CloudGraphLink with default values:
// _nodes[0] = _nodes[1] = -1
// Return NULL if couldn't create the link
CloudGraphLink* CloudGraphCreateLink(void);

// Add a copy of the link 'l' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddLink(CloudGraph *cloud, CloudGraphLink *l);

// Load the CloudGraph from 'stream'
// If 'cloud' is not NULL it is first freed
// Return 0 on success
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int CloudGraphLoad(CloudGraph **cloud, FILE *stream);

// Arrange the position of the nodes of the graph
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrange(CloudGraph *cloud, CloudGraphOpt *opt);

// Get a TGA picture representing the CloudGraph using the graphical
// options 'opt'
// Return NULL if we couldn't create the TGA
TGA* CloudGraphToTGA(CloudGraph *cloud, CloudGraphOpt *opt);

// Print the CloudGraph on 'stream'
// Do nothing if arguments are invalid
void CloudGraphPrint(CloudGraph *cloud, FILE* stream);
```

```
// Print the CloudGraphFamily on 'stream'
// Do nothing if arguments are invalid
void CloudGraphFamilyPrint(void *f, FILE *stream);

// Print the CloudGraphNode 'n' on 'stream'
// Do nothing if arguments are invalid
void CloudGraphNodePrint(void *n, FILE *stream);

// Print the CloudGraphLink on 'stream'
// Do nothing if arguments are invalid
void CloudGraphLinkPrint(void *l, FILE *stream);

// Create a new CloudGraphOpt
// Default _mode = CloudGraphModeFree
// Default _curvedLink = false
// Default _curvature = 1.0
// Default _nodeLabelMode = CloudGraphOptNodeLabelNone
// Default _familyLabelMode = CloudGraphOptFamilyLabelNone
// Default _fontSizeNode = 15
// Default _fontSizeFamily = 18
// Return NULL if we couldn't create the CloudGraphOpt
CloudGraphOpt* CloudGraphOptCreate(void);

// Free the memory used by the CloudGraphOpt
// Do nothing if arguments are invalid
void CloudGraphOptFree(CloudGraphOpt **opt);

// Set the flag defining if the links are curved to 'curved'
// Do nothing if arguments are invalid
void CloudGraphOptSetCurvedLink(CloudGraphOpt *opt, bool curved);

// Set the curvature to 'v' (in [0.0,1.0])
// Do nothing if arguments are invalid
void CloudGraphOptSetCurvature(CloudGraphOpt *opt, float v);

// Set the mode of display for nodes' label to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetNodeLabelMode(CloudGraphOpt *opt,
  CloudGraphOptNodeLabel mode);

// Set the mode of display for families' label to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetFamilyLabelMode(CloudGraphOpt *opt,
  CloudGraphOptFamilyLabel mode);

// Set the font size for nodes' label to 'size'
// Do nothing if arguments are invalid
void CloudGraphOptSetFontSizeNode(CloudGraphOpt *opt, float size);

// Set the font size for families' label to 'size'
// Do nothing if arguments are invalid
void CloudGraphOptSetFontSizeFamily(CloudGraphOpt *opt, float size);

// Return the length of the longest displayed node label
// Return 0.0 of arguments are invalid or there is no displayed label
float CloudGraphGetMaxLengthLblNode(CloudGraph *cloud, CloudGraphOpt *opt);

// Return the length of the longest displayed family label
// Return 0.0 of arguments are invalid or there is no displayed label
float CloudGraphGetMaxLengthLblFamily(CloudGraph *cloud, CloudGraphOpt *opt);

// Return the node 'id' or NULL if arguments are invalid
```

```
CloudGraphNode* CloudGraphGetNode(CloudGraph *cloud, int id);

// Return the family 'id' or NULL if arguments are invalid
CloudGraphFamily* CloudGraphGetFamily(CloudGraph *cloud, int id);

#endif
```

3 Code

```
// ============ CLOUDGRAPH.C ================

// ================= Include ==================

#include "cloudgraph.h"

// ================= Define ==================

#define rnd() (float)(rand())/(float)(RAND_MAX)
#define CLOUDGRAPH_MAXLENGTHLABEL 500

// ================ Functions declaration ====================

// Sort the nodes in the GSet in order of their families
// Do nothing if arguments are invalid
void CloudGraphSortNodeByFamily(CloudGraph *cloud);

// Arrange the position of the nodes of the graph in line
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrangeLine(CloudGraph *cloud, CloudGraphOpt *opt);

// Arrange the position of the nodes of the graph in circle
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrangeCircle(CloudGraph *cloud, CloudGraphOpt *opt);

// Update all the bounding boxes
// Do nothing if arguments are invalid
void CloudGraphUpdateBoundingBox(CloudGraph *cloud,
  CloudGraphOpt *opt);

// ================ Functions implementation ====================

// Create a new CloudGraph
// Return NULL if we couldn't create the CloudGraph
CloudGraph* CloudGraphCreate(void) {
  // Allocate memory
  CloudGraph *ret = (CloudGraph*)malloc(sizeof(CloudGraph));
  // If we could allocate memory
  if (ret != NULL) {
    // Allocate memory for the GSets
    ret->_nodes = GSetCreate();
    ret->_families = GSetCreate();
    ret->_links = GSetCreate();
    ret->_font = TGAFontCreate(tgaFontDefault);
    ret->_boundingBox = FacoidCreate(2);
    // If we couldn't allocate memory
    if (ret->_nodes == NULL || ret->_families == NULL ||
      ret->_font == NULL || ret->_boundingBox == NULL) {
```

```
      // Free memory
      CloudGraphFree(&ret);
    }
    // Set the font anchor
    TGAFontSetAnchor(ret->_font, tgaFontAnchorCenterLeft);
    // Set the font scale
    VecFloat *v = VecFloatCreate(2);
    if (v != NULL) {
      VecSet(v, 0, 0.5); VecSet(v, 1, 1.0);
      TGAFontSetScale(ret->_font, v);
      VecFree(&v);
    }
  }
  // Return the new CloudGraph
  return ret;
}

// Free the memory used by the CloudGraph
// Do nothing if arguments are invalid
void CloudGraphFree(CloudGraph **cloud) {
  // Check arguments
  if (cloud == NULL || *cloud == NULL)
    return;
  // Free memory used by nodes
  GSetElem *elem = (*cloud)->_nodes->_head;
  while (elem != NULL) {
    CloudGraphNodeFree((CloudGraphNode**)(&(elem->_data)));
    elem = elem->_next;
  }
  // Free memory used by families
  elem = (*cloud)->_families->_head;
  while (elem != NULL) {
    CloudGraphFamilyFree((CloudGraphFamily**)(&(elem->_data)));
    elem = elem->_next;
  }
  // Free memory used by links
  elem = (*cloud)->_links->_head;
  while (elem != NULL) {
    CloudGraphLinkFree((CloudGraphLink**)(&(elem->_data)));
    elem = elem->_next;
  }
  // Free memory
  GSetFree(&((*cloud)->_nodes));
  GSetFree(&((*cloud)->_families));
  GSetFree(&((*cloud)->_links));
  TGAFreeFont(&((*cloud)->_font));
  ShapoidFree(&((*cloud)->_boundingBox));
  free(*cloud);
  *cloud = NULL;
}

// Free the memory used by a CloudGraphNode
// Do nothing if arguments are invalid
void CloudGraphNodeFree(CloudGraphNode** node) {
  // Check arguments
  if (node == NULL || *node == NULL)
    return;
  // Free the memory used by the node
  VecFree(&((*node)->_pos));
  VecFree(&((*node)->_right));
  if ((*node)->_label != NULL)
    free((*node)->_label);
```

```
  ShapoidFree(&((*node)->_boundingBox));
  if ((*node)->_boundingBoxLbl != NULL)
    ShapoidFree(&((*node)->_boundingBoxLbl));
  free(*node);
  *node = NULL;
}

// Free the memory used by a CloudGraphFamily
// Do nothing if arguments are invalid
void CloudGraphFamilyFree(CloudGraphFamily** family) {
  // Check arguments
  if (family == NULL || *family == NULL)
    return;
  // Free the memory used by the family
  VecFree(&((*family)->_pos));
  VecFree(&((*family)->_right));
  if ((*family)->_label != NULL)
    free((*family)->_label);
  // Free the memory used by the bounding box
  if ((*family)->_boundingBox != NULL)
    ShapoidFree(&((*family)->_boundingBox));
  free(*family);
  *family = NULL;
}

// Free the memory used by a CloudGraphLink
// Do nothing if arguments are invalid
void CloudGraphLinkFree(CloudGraphLink** link) {
  if (link == NULL || *link == NULL)
    return;
  BCurveFree(&((*link)->_curve));
  ShapoidFree(&((*link)->_boundingBox));
  free(*link);
  *link = NULL;
}

// Set the represention mode to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetMode(CloudGraphOpt *opt, CloudGraphMode mode) {
  // Check arguments
  if (opt == NULL)
    return;
  opt->_mode = mode;
}

// Create a random CloudGraph having between 'nbNodeMin' and 'nbNodeMax'
// nodes, and between 'nbFamilyMin' and 'nbFamilyMax' families, and
// 'density' (in [0,1]) probability of connection between each pair of
// nodes, and representation mode 'mode'
// If 'cloud' is not NULL it is first freed
// The random generator must be initialized before calling this function
// Return true on success, false else (invalid arguments or malloc failed)
bool CloudGraphCreateRnd(CloudGraph **cloud, int nbNodeMin,
  int nbNodeMax, int nbFamilyMin, int nbFamilyMax, float density) {
  // Check arguments
  if (cloud == NULL || nbNodeMin < 1 || nbNodeMax < nbNodeMin ||
    nbFamilyMin < 1 || nbFamilyMax < nbFamilyMin ||
    density < 0.0 || density > 1.0)
    return false;
  // If cloud is not NULL
  if (*cloud != NULL)
    // Free the cloud
```

```
  CloudGraphFree(cloud);
// Create a new cloud
*cloud = CloudGraphCreate();
// If we couldn't create a new one
if (*cloud == NULL)
  // Stop here
  return false;
// Choose a number of nodes and families
int nbNode = nbNodeMin +
  (int)floor(rnd() * (float)(nbNodeMax - nbNodeMin));
int nbFamily = nbFamilyMin +
  (int)floor(rnd() * (float)(nbFamilyMax - nbFamilyMin));
// Declare a variable to create the families
CloudGraphFamily *family = CloudGraphCreateFamily();
// If we couldn't allocate memory
if (family == NULL) {
  // Free memory
  CloudGraphFree(cloud);
  // Stop here
  return false;
}
// Allocate memory for the label
family->_label = (char*)malloc(sizeof(char) * 100);
// If we couldn't allocate memory
if (family->_label == NULL) {
  // Stop here
  CloudGraphFamilyFree(&family);
  CloudGraphFree(cloud);
  return false;
}
// Create the families
for (int iFamily = 0; iFamily < nbFamily; ++iFamily) {
  // Set the properties
  family->_id = iFamily;
  sprintf(family->_label, "Family%03d", iFamily);
  for (int iRGB = 0; iRGB < 3; ++iRGB)
    family->_rgba[iRGB] = (char)floor(rnd() * 255.0);
  // Add the family
  bool ret = CloudGraphAddFamily(*cloud, family);
  // If we couldn't add the family
  if (ret == false) {
    // Stop here
    CloudGraphFamilyFree(&family);
    CloudGraphFree(cloud);
    return false;
  }
}
// Free memory
CloudGraphFamilyFree(&family);
// Declare a variable to create the nodes
CloudGraphNode *n = CloudGraphCreateNode();
// If we couldn't allocate memory
if (n == NULL) {
  // Stop here
  CloudGraphFree(cloud);
  return false;
}
// Create the nodes
for (int iNode = 0; iNode < nbNode; ++iNode) {
  // Set the data of the node
  n->_id = iNode;
  n->_family = (int)floor(rnd() * (float)(nbFamily));
```

```
    char label[100] = {'\0'};
    sprintf(label, "Node%03d", iNode);
    n->_label = (char*)malloc(sizeof(char) * (strlen(label) + 1));
    // If we couldn't allocate memory for the label
    if (n->_label == NULL) {
      // Stop here
      CloudGraphNodeFree(&n);
      CloudGraphFree(cloud);
      return false;
    }
    memcpy(n->_label, label, sizeof(char) * (strlen(label) + 1));
    // Add the node
    CloudGraphAddNode(*cloud, n);
    // Free memory used for the label
    free(n->_label);
    n->_label = NULL;
  }
  // Free memory
  CloudGraphNodeFree(&n);
  // Declare a variable to create the links
  CloudGraphLink *l = CloudGraphCreateLink();
  // If we couldn't allocate memory
  if (l == NULL) {
    // Stop here
    CloudGraphFree(cloud);
    return false;
  }
  // For each pair of nodes
  for (int iNode = 0; iNode < nbNode - 1; ++iNode) {
    for (int jNode = iNode + 1; jNode < nbNode; ++jNode) {
      // If the link between this pair exist
      if (rnd() <= density) {
        // Set the data
        l->_nodes[0] = iNode;
        l->_nodes[1] = jNode;
        // Add the link
        bool ret = CloudGraphAddLink(*cloud, l);
        // If we couldn't add the link
        if (ret == false) {
          // Free memory
          CloudGraphLinkFree(&l);
          CloudGraphFree(cloud);
          // Stop here
          return false;
        }
      }
    }
  }
  // Free memory
  CloudGraphLinkFree(&l);
  // Return the success code
  return true;
}

// Create a CloudGraphFamily with default values:
// _id = 0;
// _rgba = {0, 0, 0, 255}
// _label = NULL;
// Return NULL if couldn't create the family
CloudGraphFamily* CloudGraphCreateFamily(void) {
  // Allocate memory
  CloudGraphFamily *ret =
```

```
    (CloudGraphFamily*)malloc(sizeof(CloudGraphFamily));
  // If we could allocate memory
  if (ret != NULL) {
    ret->_pos = VecFloatCreate(2);
    ret->_right = VecFloatCreate(2);
    if (ret->_pos == NULL || ret->_right == NULL) {
      VecFree(&(ret->_pos));
      VecFree(&(ret->_right));
      free(ret);
      return NULL;
    }
    ret->_id = 0;
    ret->_rgba[0] = ret->_rgba[1] = ret->_rgba[2] = 0;
    ret->_rgba[3] = 255;
    ret->_label = NULL;
    ret->_boundingBox = NULL;
  }
  return ret;
}

// Add a copy of the family 'f' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddFamily(CloudGraph *cloud, CloudGraphFamily *f) {
  // Check arguments
  if (cloud == NULL || f == NULL || cloud->_families == NULL)
    return false;
  // Check that this family doesn't exist yet
  GSetElem *ptr = cloud->_families->_head;
  while (ptr != NULL) {
    // If the ID of the family to add is already used
    if (f->_id == ((CloudGraphFamily*)(ptr->_data))->_id)
      // Stop here
      return false;
    ptr = ptr->_next;
  }
  // Allocate memory for the copy of the family
  CloudGraphFamily *family = CloudGraphCreateFamily();
  // If we couldn't allocate memory
  if (family == NULL)
    // Stop here
    return false;
  // Copy the data
  family->_id = f->_id;
  for (int iRGB = 4; iRGB--;)
    family->_rgba[iRGB] = f->_rgba[iRGB];
  VecCopy(family->_pos, f->_pos);
  VecCopy(family->_right, f->_right);
  // If there is a label
  if (f->_label != NULL) {
    // Allocate memory for the copy of the label
    family->_label =
      (char*)malloc(sizeof(char) * (1 + strlen(f->_label)));
    // If we couldn't allocate memory
    if (family->_label == NULL) {
      //Free memory
      free(family);
      // Stop here
      return false;
    }
    // Copy the label
    memcpy(family->_label, f->_label,
```

```
      sizeof(char) * (1 + strlen(f->_label)));
  }
  // If there is a boundig box
  ShapoidFree(&(family->_boundingBox));
  if (f->_boundingBox != NULL) {
    family->_boundingBox = ShapoidClone(f->_boundingBox);
    if (family->_boundingBox == NULL) {
      CloudGraphFamilyFree(&family);
      return false;
    }
  }
  // Add the family to the GSet
  GSetAppend(cloud->_families, family);
  // Return the success code
  return true;
}

// Create a CloudGraphNode with default values:
// _id = 0
// _family = 0
// _label = NULL
// Return NULL if couldn't create the family
CloudGraphNode* CloudGraphCreateNode(void) {
  // Allocate memory
  CloudGraphNode *ret = (CloudGraphNode*)malloc(sizeof(CloudGraphNode));
  // If we could allocate memory
  if (ret != NULL) {
    ret->_pos = VecFloatCreate(2);
    ret->_right = VecFloatCreate(2);
    ret->_boundingBox = FacoidCreate(2);
    if (ret->_pos == NULL || ret->_right == NULL ||
      ret->_boundingBox == NULL) {
      VecFree(&(ret->_pos));
      VecFree(&(ret->_right));
      ShapoidFree(&(ret->_boundingBox));
      free(ret);
      return NULL;
    }
    ret->_id = 0;
    ret->_family = 0;
    ret->_label = NULL;
    ret->_boundingBoxLbl = NULL;
  }
  return ret;
}

// Add a copy of the node 'n' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddNode(CloudGraph *cloud, CloudGraphNode *n) {
  // Check arguments
  if (cloud == NULL || n == NULL)
    return false;
  // Create the node to add
  CloudGraphNode *node = CloudGraphCreateNode();
  // If we couldn't allocate memory
  if (node == NULL)
    // Stop here
    return false;
  // Copy the data of the node
  node->_id = n->_id;
  node->_family = n->_family;
```

```
  VecCopy(node->_pos, n->_pos);
  VecCopy(node->_right, n->_right);
  // If there is a label
  if (n->_label != NULL) {
    // Allocate memory for the label of the node
    node->_label =
      (char*)malloc(sizeof(char) * (strlen(n->_label) + 1));
    // If we couldn't allocate memory
    if (node->_label == NULL) {
      // Free memory
      CloudGraphNodeFree(&node);
      // Stop here
      return false;
    }
    // Copy the label
    memcpy(node->_label, n->_label,
      sizeof(char) * (strlen(n->_label) + 1));
  }
  ShapoidFree(&(node->_boundingBox));
  node->_boundingBox = ShapoidClone(n->_boundingBox);
  if (node->_boundingBox == NULL) {
    CloudGraphNodeFree(&node);
    return false;
  }
  // If there is a bounding box for the label
  if (n->_boundingBoxLbl != NULL) {
    node->_boundingBoxLbl = ShapoidClone(n->_boundingBoxLbl);
    if (node->_boundingBoxLbl == NULL) {
      CloudGraphNodeFree(&node);
      return false;
    }
  }
  // Add the node to the set
  GSetAppend(cloud->_nodes, node);
  // Return success code
  return true;
}

// Create a CloudGraphLink with default values:
// _nodes[0] = _nodes[1] = -1
// Return NULL if couldn't create the link
CloudGraphLink* CloudGraphCreateLink(void) {
  // Allocate memory
  CloudGraphLink *ret = (CloudGraphLink*)malloc(sizeof(CloudGraphLink));
  // If we could allocate memory
  if (ret != NULL) {
    // Set the properties
    ret->_nodes[0] = ret->_nodes[1] = -1;
    ret->_boundingBox = NULL;
    // Create the curve
    ret->_curve = BCurveCreate(3, 2);
    if (ret->_curve == NULL)
      CloudGraphLinkFree(&ret);
  }
  return ret;
}

// Add a copy of the link 'l' to the CloudGraph
// Return false if the arguments are invalid or memory allocation failed
// else return true
bool CloudGraphAddLink(CloudGraph *cloud, CloudGraphLink *l) {
  // Check arguments
```

```
  if (cloud == NULL || l == NULL || cloud->_links == NULL ||
    l->_nodes[0] == l->_nodes[1])
    return false;
  // Allocate memory for the copy of the link
  CloudGraphLink *link = CloudGraphCreateLink();
  // If we couldn't allocate memory
  if (link == NULL)
    // Stop here
    return false;
  // Copy the data
  for (int iNode = 2; iNode--;) {
    link->_nodes[iNode] = l->_nodes[iNode];
    link->_families[iNode] = l->_families[iNode];
  }
  BCurveFree(&(link->_curve));
  link->_curve = BCurveClone(l->_curve);
  if (link->_curve == NULL) {
    CloudGraphLinkFree(&link);
    return false;
  }
  // Add the link to the set
  GSetAppend(cloud->_links, link);
  // Return success code
  return true;
}

// Load the CloudGraph from 'stream'
// If 'cloud' is not NULL it is first freed
// Return 0 on success
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int CloudGraphLoad(CloudGraph **cloud, FILE *stream) {
  // Check arguments
  if (*cloud == NULL || stream == NULL)
    return 1;
  // If cloud already exists
  if (*cloud != NULL)
    // Free it
    CloudGraphFree(cloud);
  // Create the cloud
  *cloud = CloudGraphCreate();
  // Create a family
  CloudGraphFamily *family = CloudGraphCreateFamily();
  // Create a node
  CloudGraphNode *node = CloudGraphCreateNode();
  // Create a link
  CloudGraphLink *link = CloudGraphCreateLink();
  // If we couldn't allocate memory
  if (*cloud == NULL || family == NULL || node == NULL ||
    link == NULL) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 2;
  }
  // Set the opacity of the family color
  family->_rgba[3] = 255;
  // Allocate memory for the family and node label
```

```
// The local family and node are just used to here as an argument
// the proper amount of memory will be allocated when they are
// added to the cloud
family->_label =
  (char*)malloc(sizeof(char) * CLOUDGRAPH_MAXLENGTHLABEL);
node->_label =
  (char*)malloc(sizeof(char) * CLOUDGRAPH_MAXLENGTHLABEL);
if (family->_label == NULL || node->_label == NULL) {
  // Free memory and stop here
  CloudGraphFree(cloud);
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  return 2;
}
// Declare a variable to read the rgb
int rgb[3] = {0};
// Read the number of families
int nbFamily = 0;
int ret = fscanf(stream, "%d", &nbFamily);
// If the fscanf failed
if (ret == EOF) {
  // Free memory and stop here
  CloudGraphFree(cloud);
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  return 4;
}
// If the number of family is invalid
if (nbFamily <= 0) {
  // Free memory and stop here
  CloudGraphFree(cloud);
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  return 3;
}
// For each family
for (int iFamily = nbFamily; iFamily--;) {
  // Read the family properties
  ret = fscanf(stream, "%d %d %d %d ",
    &(family->_id), rgb, rgb + 1, rgb + 2);
  // Check values
  if (ret == EOF || family->_id < 0 || family->_id >= nbFamily ||
    rgb[0] < 0 || rgb[0] > 255 || rgb[1] < 0 || rgb[1] > 255 ||
    rgb[2] < 0 || rgb[2] > 255) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
  }
  if (fgets(family->_label, CLOUDGRAPH_MAXLENGTHLABEL,
    stream) == NULL) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
```

```
  }
  // Remove the line return
  family->_label[strlen(family->_label) - 1] = '\0';
  // Convert rgb values
  for (int iRgb = 3; iRgb--;)
    family->_rgba[iRgb] = rgb[iRgb];
  // Add the family to the cloud
  if (CloudGraphAddFamily(*cloud, family) == false) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
  }
}
// Read the number of nodes
int nbNode = 0;
ret = fscanf(stream, "%d", &nbNode);
// If the fscanf failed
if (ret == EOF) {
  // Free memory and stop here
  CloudGraphFree(cloud);
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  return 4;
}
// If the number of node is invalid
if (nbNode <= 0) {
  // Free memory and stop here
  CloudGraphFree(cloud);
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  return 3;
}
// For each node
for (int iNode = nbNode; iNode--;) {
  // Read the node properties
  ret = fscanf(stream, "%d %d ",
    &(node->_id), &(node->_family));
  // Check values
  if (ret == EOF || node->_id < 0 || node->_id >= nbNode ||
    node->_family < 0 || node->_family >= nbFamily) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
  }
  if (fgets(node->_label, CLOUDGRAPH_MAXLENGTHLABEL,
    stream) == NULL) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
  }
  // Remove the line return
```

```
    node->_label[strlen(node->_label) - 1] = '\0';
    // Add the node to the cloud
    if (CloudGraphAddNode(*cloud, node) == false) {
      // Free memory and stop here
      CloudGraphFree(cloud);
      CloudGraphFamilyFree(&family);
      CloudGraphNodeFree(&node);
      CloudGraphLinkFree(&link);
      return 3;
    }
  }
  // Read the number of link
  int nbLink;
  ret = fscanf(stream, "%d", &nbLink);
  // If the fscanf failed
  if (ret == EOF) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 4;
  }
  // If the number of link is invalid
  if (nbLink < 0) {
    // Free memory and stop here
    CloudGraphFree(cloud);
    CloudGraphFamilyFree(&family);
    CloudGraphNodeFree(&node);
    CloudGraphLinkFree(&link);
    return 3;
  }
  // For each node
  for (int iLink = nbLink; iLink--;) {
    // Read the link properties
    ret = fscanf(stream, "%d %d", link->_nodes, link->_nodes + 1);
    // Check values
    if (link->_nodes[0] < 0 || link->_nodes[0] >= nbNode ||
      link->_nodes[1] < 0 || link->_nodes[1] >= nbNode) {
      // Free memory and stop here
      CloudGraphFree(cloud);
      CloudGraphFamilyFree(&family);
      CloudGraphNodeFree(&node);
      CloudGraphLinkFree(&link);
      return 3;
    }
    if (CloudGraphAddLink(*cloud, link) == false) {
      // Free memory and stop here
      CloudGraphFree(cloud);
      CloudGraphFamilyFree(&family);
      CloudGraphNodeFree(&node);
      CloudGraphLinkFree(&link);
      return 3;
    }
  }
  // Free memory
  CloudGraphFamilyFree(&family);
  CloudGraphNodeFree(&node);
  CloudGraphLinkFree(&link);
  // Return the success code
  return 0;
}
```

```
// Sort the masses in the GSet of the SpringSys in order of their
// families
// Do nothing if arguments are invalid
void CloudGraphSortNodeByFamily(CloudGraph *cloud) {
  // Check arguments
  if (cloud == NULL)
    return;
  // Declare a pointer to the first element of the GSet of nodes
  GSetElem *elem = cloud->_nodes->_head;
  // Loop over the masses
  while (elem != NULL) {
    // Set the sort value of this element to the family index of its
    // node
    CloudGraphNode *node = (CloudGraphNode*)(elem->_data);
    elem->_sortVal = (float)(node->_family);
    // Move the pointer to the next mass
    elem = elem->_next;
  }
  // Sort the GSet of masses
  GSetSort(cloud->_nodes);
}

// Arrange the position of the nodes of the graph in line
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrangeLine(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return false;
  // Declare a variable to calculate the position of families label
  float *posFamily =
    (float*)malloc(sizeof(float) * cloud->_families->_nbElem);
  int *nbFamily =
    (int*)malloc(sizeof(int) * cloud->_families->_nbElem);
  // If we couldn't allocate memory
  if (posFamily == NULL || nbFamily == NULL) {
    if (posFamily != NULL) free(posFamily);
    if (nbFamily != NULL) free(nbFamily);
    // Stop here
    return false;
  }
  for (int iFamily = cloud->_families->_nbElem; iFamily--;) {
    posFamily[iFamily] = 0.0;
    nbFamily[iFamily] = 0;
  }
  // Declare a pointer toward the nodes
  GSetElem *ptr = cloud->_nodes->_head;
  // Declare a variable to memorize the index of the node in the set
  int iNode = 0;
  // Loop on the nodes
  while (ptr != NULL) {
    // Declare a pointer to the node
    CloudGraphNode *node = (CloudGraphNode*)(ptr->_data);
    // Set the position of the node
    VecSet(node->_pos, 0, 0.0);
    VecSet(node->_pos, 1,
      2.0 * opt->_fontSizeNode * ((float)iNode + 0.5));
    // Calculate the family position
    posFamily[node->_family] += VecGet(node->_pos, 1);
    ++(nbFamily[node->_family]);
    // Set the right of the node
```

```
    VecSet(node->_right, 0, 1.0);
    VecSet(node->_right, 1, 0.0);
    // Set the angle with abciss
    node->_theta = 0.0;
    // Continue with the next node
    ptr = ptr->_next;
    // Increment the index of the node
    ++iNode;
  }
  // Calculate the family position
  for (int iFamily = cloud->_families->_nbElem; iFamily--;)
    if (nbFamily[iFamily] != 0)
      posFamily[iFamily] /= (float)(nbFamily[iFamily]);
  // Set the pointer to the head of the set of links
  ptr = cloud->_links->_head;
  // Loop on the links
  while (ptr != NULL) {
    CloudGraphLink *link = (CloudGraphLink*)(ptr->_data);
    // Get the two nodes of this link
    CloudGraphNode *nodes[2];
    for (int iNode = 2; iNode--;)
      nodes[iNode] = CloudGraphGetNode(cloud, link->_nodes[iNode]);
    if (nodes[0] != NULL && nodes[1] != NULL) {
      // Set the values of the BCurve for this link
      VecCopy(link->_curve->_ctrl[0], nodes[0]->_pos);
      VecCopy(link->_curve->_ctrl[1], nodes[0]->_pos);
      VecCopy(link->_curve->_ctrl[2], nodes[1]->_pos);
      VecCopy(link->_curve->_ctrl[3], nodes[1]->_pos);
      float dist = VecDist(nodes[0]->_pos, nodes[1]->_pos);
      VecOp(link->_curve->_ctrl[1], 1.0, nodes[0]->_right,
        -1.0 * dist * opt->_curvature);
      VecOp(link->_curve->_ctrl[2], 1.0, nodes[1]->_right,
        -1.0 * dist * opt->_curvature);
      // Memorize the family of each node
      for (int iNode = 2; iNode--;)
        link->_families[iNode] = nodes[iNode]->_family;
    }
    // Move to next link
    ptr = ptr->_next;
  }
  // Set the pointer to the head of the set of families
  ptr = cloud->_families->_head;
  // Loop on the families
  while (ptr != NULL) {
    CloudGraphFamily *family = (CloudGraphFamily*)(ptr->_data);
    // Set the position
    VecSet(family->_pos, 0, opt->_fontSizeNode);
    VecSet(family->_pos, 1, posFamily[family->_id]);
    // Set the right direction
    VecSet(family->_right, 0, 1.0);
    VecSet(family->_right, 1, 0.0);
    // Move to next family
    ptr = ptr->_next;
  }
  // Free memory
  free(posFamily);
  free(nbFamily);
  // Return success code
  return true;
}

// Arrange the position of the nodes of the graph in circle
```

```
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrangeCircle(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return false;
  // Declare a variable to calculate the position of families label
  float *posFamily =
    (float*)malloc(sizeof(float) * cloud->_families->_nbElem);
  int *nbFamily =
    (int*)malloc(sizeof(int) * cloud->_families->_nbElem);
  // If we couldn't allocate memory
  if (posFamily == NULL || nbFamily == NULL) {
    if (posFamily != NULL) free(posFamily);
    if (nbFamily != NULL) free(nbFamily);
    // Stop here
    return false;
  }
  for (int iFamily = cloud->_families->_nbElem; iFamily--;) {
    posFamily[iFamily] = 0.0;
    nbFamily[iFamily] = 0;
  }
  // Declare a variable to memorize the radius of the circle
  float r = (float)(cloud->_nodes->_nbElem) *
    opt->_fontSizeNode / PBMATH_PI;
  // Declare a pointer toward the nodes
  GSetElem *ptr = cloud->_nodes->_head;
  // Declare variables to position the nodes
  float theta = 0.0;
  float dTheta = 2.0 * PBMATH_PI / (float)(cloud->_nodes->_nbElem);
  // Loop on the nodes
  while (ptr != NULL) {
    // Declare a pointer to the node
    CloudGraphNode *node = (CloudGraphNode*)(ptr->_data);
    // Set the position of the node
    VecSet(node->_pos, 0, r * cos(theta));
    VecSet(node->_pos, 1, r * sin(theta));
    // Set the right of the node
    VecSet(node->_right, 0, 1.0);
    VecSet(node->_right, 1, 0.0);
    VecRot2D(node->_right, theta);
    // Set the angle with abciss
    node->_theta = theta;
    // Calculate the family position
    posFamily[node->_family] += theta;
    ++(nbFamily[node->_family]);
    // Continue with the next node
    ptr = ptr->_next;
    // Increment the angle
    theta += dTheta;
  }
  // Calculate the family position
  for (int iFamily = cloud->_families->_nbElem; iFamily--;)
    if (nbFamily[iFamily] != 0)
      posFamily[iFamily] /= (float)(nbFamily[iFamily]);
  // Set the pointer to the head of the set of links
  ptr = cloud->_links->_head;
  // Loop on the links
  while (ptr != NULL) {
    CloudGraphLink *link = (CloudGraphLink*)(ptr->_data);
    // Get the two nodes of this link
    CloudGraphNode *nodes[2];
```

```
    for (int iNode = 2; iNode--;)
      nodes[iNode] = CloudGraphGetNode(cloud, link->_nodes[iNode]);
    if (nodes[0] != NULL && nodes[1] != NULL) {
      // Set the values of the BCurve for this link
      VecCopy(link->_curve->_ctrl[0], nodes[0]->_pos);
      VecCopy(link->_curve->_ctrl[1], nodes[0]->_pos);
      VecCopy(link->_curve->_ctrl[2], nodes[1]->_pos);
      VecCopy(link->_curve->_ctrl[3], nodes[1]->_pos);
      float dist = VecDist(nodes[0]->_pos, nodes[1]->_pos);
      VecOp(link->_curve->_ctrl[1], 1.0, nodes[0]->_right,
        -1.0 * dist * opt->_curvature * 0.5);
      VecOp(link->_curve->_ctrl[2], 1.0, nodes[1]->_right,
        -1.0 * dist * opt->_curvature * 0.5);
      // Memorize the family of each node
      for (int iNode = 2; iNode--;)
        link->_families[iNode] = nodes[iNode]->_family;
    }
    // Move to next link
    ptr = ptr->_next;
  }
  // Set the pointer to the head of the set of families
  ptr = cloud->_families->_head;
  // Loop on the families
  while (ptr != NULL) {
    CloudGraphFamily *family = (CloudGraphFamily*)(ptr->_data);
    // Set the position
    VecSet(family->_pos, 0, r + opt->_fontSizeNode);
    VecSet(family->_pos, 1, 0.0);
    VecRot2D(family->_pos, posFamily[family->_id]);
    // Set the right direction
    VecSet(family->_right, 0, 1.0);
    VecSet(family->_right, 1, 0.0);
    VecRot2D(family->_right, posFamily[family->_id]);
    // Move to next family
    ptr = ptr->_next;
  }
  // Free memory
  free(posFamily);
  free(nbFamily);
  // Return success code
  return true;
}

// Arrange the position of the nodes of the graph
// Return true if it could arrange nodes
// Return false if arguments are invalid or it couldn't arrange nodes
bool CloudGraphArrange(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return false;
  // Ensure the nodes are ordered by family
  CloudGraphSortNodeByFamily(cloud);
  // Declare a variable for the return value
  bool ret = true;
  // Set initial position of nodes depending on representation mode
  // If the representation is circle or free
  if (opt->_mode == CloudGraphModeCircle) {
    ret = CloudGraphArrangeCircle(cloud, opt);
  } else if (opt->_mode == CloudGraphModeLine) {
    ret = CloudGraphArrangeLine(cloud, opt);
  }
  // Update the bounding boxes of nodes and families' labels
```

```
  CloudGraphUpdateBoundingBox(cloud, opt);
  // Return the success value
  return ret;
}

// Get a TGA picture representing the CloudGraph using the graphical
// options 'opt'
// Return NULL if we couldn't create the TGA
TGA* CloudGraphToTGA(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return NULL;
  // Declare a variable for the returned TGA
  TGA *tga = NULL;
  // Declare a variable to memorize the size of nodes
  VecFloat *sizeNode = VecFloatCreate(2);
  // Declare a vector to calculate positions
  VecFloat *pos = VecFloatCreate(2);
  // Create a default pencil
  TGAPencil *pen = TGAGetPencil();
  // Create a pixel for drawing
  TGAPixel *pixel = TGAGetWhitePixel();
  // Declare a variable to memorize the dimensions of the tga
  VecShort *dim = VecShortCreate(2);
  // Declare a variable to memorize empty family
  bool *emptyFamily =
    (bool*)malloc(sizeof(bool) * cloud->_families->_nbElem);
  // If we couldn't allocate memory
  if (pos == NULL || sizeNode == NULL || pen == NULL || pixel == NULL ||
    dim == NULL || emptyFamily == NULL) {
    // Free memory and stop here
    VecFree(&pos);
    VecFree(&sizeNode);
    VecFree(&dim);
    TGAPixelFree(&pixel);
    TGAPencilFree(&pen);
    if (emptyFamily != NULL) free(emptyFamily);
    return NULL;
  }
  // Set the family to empty by default
  for (int iFamily = cloud->_families->_nbElem; iFamily--;)
    emptyFamily[iFamily] = true;
  // Set the dimension of the tga
  for (int i = 2; i--;)
    VecSet(dim, i,
      (short)floor(VecGet(cloud->_boundingBox->_axis[i], i)));
  // Set the size of the node, it's equal to the size of the font
  for (int i = 2; i--;)
    VecSet(sizeNode, i, 0.5 * opt->_fontSizeNode);
  // Create the TGA
  tga = TGACreate(dim, pixel);
  // If we couldn't create the tga
  if (tga == NULL) {
    // Free memory and stop here
    VecFree(&pos);
    VecFree(&sizeNode);
    TGAPixelFree(&pixel);
    TGAPencilFree(&pen);
    free(emptyFamily);
    return NULL;
  }
  // Set the pen properties
```

```
TGAPencilSetShapeRound(pen);
TGAPencilSetAntialias(pen, true);
TGAPencilSetThickness(pen, 2.0);
// Set the font size
TGAFontSetSize(cloud->_font, opt->_fontSizeNode);
// Declare a pointer toward the nodes
GSetElem *ptr = cloud->_nodes->_head;
// Loop on the nodes
while (ptr != NULL) {
  // Declare a pointer to the node
  CloudGraphNode *node = (CloudGraphNode*)(ptr->_data);
  // Update family emptiness
  emptyFamily[node->_family] = false;
  // Declare a pointer to the family of the node
  CloudGraphFamily *family = GSetGet(cloud->_families,
    node->_family);
  // If we could get the family
  if (family != NULL) {
    // Set the color of the pencil to the color of the family
    TGAPencilSetColRGBA(pen, family->_rgba);
    // Draw the node
    VecCopy(pos, node->_pos);
    VecOp(pos, 1.0, cloud->_boundingBox->_pos, -1.0);
    TGAFillEllipse(tga, pos, sizeNode, pen);
    // If this node label must be displayed
    if (opt->_nodeLabelMode == CloudGraphOptNodeLabelAll) {
      // Set the position for the label string
      VecCopy(pos, node->_boundingBoxLbl->_pos);
      VecOp(pos, 1.0, cloud->_boundingBox->_pos, -1.0);
      // Set the angle of the font
      TGAFontSetRight(cloud->_font, node->_right);
      // Draw the string
      TGAPrintString(tga, pen, cloud->_font,
        (unsigned char*)(node->_label), pos);
    }
  }
  // Move to next node
  ptr = ptr->_next;
}
// Set the pointer to the head of the set of links
ptr = cloud->_links->_head;
// Set the pen mode
TGAPencilSetModeColorBlend(pen, 0, 1);
// Loop on the links
while (ptr != NULL) {
  CloudGraphLink *link = (CloudGraphLink*)(ptr->_data);
  // Set the colors
  for (int iNode = 2; iNode--;) {
    CloudGraphFamily *family =
      CloudGraphGetFamily(cloud, link->_families[iNode]);
    TGAPencilSelectColor(pen, iNode);
    TGAPencilSetColRGBA(pen, family->_rgba);
  }
  // Translate the curve to its position
  VecOp(cloud->_boundingBox->_pos, -1.0, NULL, 0.0);
  BCurveTranslate(link->_curve, cloud->_boundingBox->_pos);
  VecOp(cloud->_boundingBox->_pos, -1.0, NULL, 0.0);
  // Draw the link
  TGADrawCurve(tga, link->_curve, pen);
  // Put back the curve to it original position
  BCurveTranslate(link->_curve, cloud->_boundingBox->_pos);
  // Move to next link
```

```
    ptr = ptr->_next;
  }
  // If the families label must be displayed
  if (opt->_familyLabelMode == CloudGraphOptFamilyLabelAll) {
    // Set the pen mode
    TGAPencilSetModeColorSolid(pen);
    // Set the pointer to the head of the set of families
    ptr = cloud->_families->_head;
    // Loop on the families
    while (ptr != NULL) {
      CloudGraphFamily *family = (CloudGraphFamily*)(ptr->_data);
      // If this family is not empty
      if (emptyFamily[family->_id] == false) {
        // Set the color
        TGAPencilSetColRGBA(pen, family->_rgba);
        // Set the angle of the font
        TGAFontSetRight(cloud->_font, family->_right);
        // Set the position
        VecCopy(pos, family->_pos);
        VecOp(pos, 1.0, cloud->_boundingBox->_pos, -1.0);
        // Draw the string
        TGAPrintString(tga, pen, cloud->_font,
          (unsigned char*)(family->_label), pos);
      }
      // Move to next family
      ptr = ptr->_next;
    }
  }
  // Free memory
  VecFree(&pos);
  VecFree(&sizeNode);
  TGAPixelFree(&pixel);
  TGAPencilFree(&pen);
  VecFree(&dim);
  free(emptyFamily);
  // Return the TGA
  return tga;
}

// Update all the bounding boxes
// Do nothing if arguments are invalid
void CloudGraphUpdateBoundingBox(CloudGraph *cloud,
  CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return;
  // Declare a variable to create the set of all bounding boxes
  GSet *set = GSetCreate();
  // If we couldn't allocate memory
  if (set == NULL) {
    return;
  }
  // Declare a pointer to go through the sets
  GSetElem *ptr = cloud->_nodes->_head;
  // Set the size of the font to the node font size
  TGAFontSetSize(cloud->_font, opt->_fontSizeNode);
  //Declare a variable to memorize the length of longest label
  float maxLength = 0.0;
  // Loop through nodes
  while (ptr != NULL) {
    CloudGraphNode *node = (CloudGraphNode*)(ptr->_data);
    // Update the bounding box for the node
```

```
  VecCopy(node->_boundingBox->_pos, node->_pos);
  VecSet(node->_boundingBox->_axis[0], 0, opt->_fontSizeNode);
  VecSet(node->_boundingBox->_axis[0], 1, 0.0);
  VecSet(node->_boundingBox->_axis[1], 0, 0.0);
  VecSet(node->_boundingBox->_axis[1], 1, opt->_fontSizeNode);
  VecOp(node->_boundingBox->_pos, 1.0,
    node->_boundingBox->_axis[0], -0.5);
  VecOp(node->_boundingBox->_pos, 1.0,
    node->_boundingBox->_axis[1], -0.5);
  // Create the bounding box for the label
  if (node->_boundingBoxLbl != NULL)
    ShapoidFree(&(node->_boundingBoxLbl));
  TGAFontSetRight(cloud->_font, node->_right);
  node->_boundingBoxLbl = TGAFontGetStringBound(cloud->_font,
    (unsigned char*)(node->_label));
  // Update the length of longest label
  float l = VecNorm(node->_boundingBoxLbl->_axis[0]);
  if (l > maxLength)
    maxLength = l;
  // Place the bounding box for the label
  VecCopy(node->_boundingBoxLbl->_pos, node->_pos);
  VecOp(node->_boundingBoxLbl->_pos, 1.0,
    node->_right, opt->_fontSizeNode);
  // Add the bounding box to the set
  GSetAppend(set, node->_boundingBox);
  // if the node labels are displayed
  if (opt->_nodeLabelMode != CloudGraphOptNodeLabelNone)
    // Add the label's bounding box to the set
    GSetAppend(set, node->_boundingBoxLbl);
  // Move to the next node
  ptr = ptr->_next;
}
// Set the pointer to the head of the set of links
ptr = cloud->_links->_head;
// Loop through the links
while (ptr != NULL) {
  CloudGraphLink *link = (CloudGraphLink*)(ptr->_data);
  // Create the bounding box
  if (link->_boundingBox != NULL)
    ShapoidFree(&(link->_boundingBox));
  link->_boundingBox = BCurveGetBoundingBox(link->_curve);
  // Add the bounding box to the set
  GSetAppend(set, link->_boundingBox);
  // Move to the next link
  ptr = ptr->_next;
}
// Set the pointer to the head of the set of families
ptr = cloud->_families->_head;
// Loop through the families
while (ptr != NULL) {
  CloudGraphFamily *family = (CloudGraphFamily*)(ptr->_data);
  // Create the bounding box for the label
  if (family->_boundingBox != NULL)
    ShapoidFree(&(family->_boundingBox));
  TGAFontSetRight(cloud->_font, family->_right);
  family->_boundingBox = TGAFontGetStringBound(cloud->_font,
    (unsigned char*)(family->_label));
  // If the node labels are displayed
  if (opt->_nodeLabelMode != CloudGraphOptNodeLabelNone)
    // Correct the position of the family label
    VecOp(family->_pos, 1.0, family->_right,
      maxLength + opt->_fontSizeNode);
```

```
    // Place the bounding box for the label
    VecCopy(family->_boundingBox->_pos, family->_pos);
    // If the family labels are displayed
    if (opt->_familyLabelMode != CloudGraphOptFamilyLabelNone)
      // Add the label's bounding box to the set
      GSetAppend(set, family->_boundingBox);
    // Move to the next family
    ptr = ptr->_next;
  }
  // Create the whole bounding box
  if (cloud->_boundingBox != NULL)
    ShapoidFree(&(cloud->_boundingBox));
  cloud->_boundingBox = ShapoidGetBoundingBox(set);
  // Free the set
  GSetFree(&set);
  // Add some pixels to the border
  for (int iDim = 2; iDim--;) {
    VecSet(cloud->_boundingBox->_pos, iDim,
      VecGet(cloud->_boundingBox->_pos, iDim) - opt->_fontSizeNode);
    VecSet(cloud->_boundingBox->_axis[iDim], iDim,
      VecGet(cloud->_boundingBox->_axis[iDim], iDim) +
      opt->_fontSizeNode * 2.0);
  }
}

// Print the CloudGraph on 'stream'
// Do nothing if arguments are invalid
void CloudGraphPrint(CloudGraph *cloud, FILE* stream) {
  // Check arguments
  if (cloud == NULL || stream == NULL)
    return;
  // Print the families
  fprintf(stream, "Families:\n");
  GSetPrint(cloud->_families, stream,
    &CloudGraphFamilyPrint, (char*)"\n");
  fprintf(stream, "\n");
  // Print the nodes
  fprintf(stream, "Nodes:\n");
  GSetPrint(cloud->_nodes, stream,
    &CloudGraphNodePrint, (char*)"\n");
  fprintf(stream, "\n");
  // Print the links
  fprintf(stream, "Links:\n");
  GSetPrint(cloud->_links, stream,
    &CloudGraphLinkPrint, (char*)"\n");
  fprintf(stream, "\n");

}

// Print the CloudGraphFamily on 'stream'
// Do nothing if arguments are invalid
void CloudGraphFamilyPrint(void *f, FILE *stream) {
  // Check arguments
  if (f == NULL || stream == NULL)
    return;
  // Print the family's properties
  fprintf(stream, "#%d rgb(%03d,%03d,%03d)",
    ((CloudGraphFamily*)f)->_id,
    ((CloudGraphFamily*)f)->_rgba[0],
    ((CloudGraphFamily*)f)->_rgba[1],
    ((CloudGraphFamily*)f)->_rgba[2]);
  if (((CloudGraphFamily*)f)->_label != NULL)
```

```
    fprintf(stream, " %s", ((CloudGraphFamily*)f)->_label);
}

// Print the CloudGraphNode 'n' on 'stream'
// Do nothing if arguments are invalid
void CloudGraphNodePrint(void *n, FILE *stream) {
  // Check arguments
  if (n == NULL || stream == NULL)
    return;
  // Print the node's properties
  fprintf(stream, "#%d family(%d) ",
    ((CloudGraphNode*)n)->_id, ((CloudGraphNode*)n)->_family);
  VecPrint(((CloudGraphNode*)n)->_pos, stream);
  if (((CloudGraphNode*)n)->_label != NULL)
    fprintf(stream, " %s", ((CloudGraphNode*)n)->_label);
}

// Print the CloudGraphLink on 'stream'
// Do nothing if arguments are invalid
void CloudGraphLinkPrint(void *l, FILE *stream) {
  // Check arguments
  if (l == NULL || stream == NULL)
    return;
  // Print the link's properties
  fprintf(stream, "%03d-%03d",
    ((CloudGraphLink*)l)->_nodes[0],
    ((CloudGraphLink*)l)->_nodes[1]);
}

// Create a new CloudGraphOpt
// Default _mode = CloudGraphModeFree
// Default _curvedLink = false
// Default _nodeLabelMode = CloudGraphOptNodeLabelNone
// Default _familyLabelMode = CloudGraphOptFamilyLabelNone
// Default _fontSizeNode = 18
// Default _fontSizeFamily = 22
// Return NULL if we couldn't create the CloudGraphOpt
CloudGraphOpt* CloudGraphOptCreate(void) {
  // Allocate memory
  CloudGraphOpt *ret  = (CloudGraphOpt*)malloc(sizeof(CloudGraphOpt));
  // If we could allocate memory
  if (ret != NULL) {
    ret->_mode = CloudGraphModeLine;
    ret->_curvedLink = false;
    ret->_curvature = 1.0;
    ret->_nodeLabelMode = CloudGraphOptNodeLabelNone;
    ret->_familyLabelMode = CloudGraphOptFamilyLabelNone;
    ret->_fontSizeNode = 18;
    ret->_fontSizeFamily = 22;
  }
  return ret;
}

// Free the memory used by the CloudGraphOpt
// Do nothing if arguments are invalid
void CloudGraphOptFree(CloudGraphOpt **opt) {
  // Check arguments
  if (opt == NULL || *opt == NULL)
    return;
  // Free memory
  free(*opt);
  *opt = NULL;
```

```
}

// Set the flag defining if the links are curved to 'curved'
// Do nothing if arguments are invalid
void CloudGraphOptSetCurvedLink(CloudGraphOpt *opt, bool curved) {
  // Check arguments
  if (opt == NULL)
    return;
  // Set the mode
  opt->_curvedLink = curved;
}

// Set the curvature to 'v' (in [0.0,1.0])
// Do nothing if arguments are invalid
void CloudGraphOptSetCurvature(CloudGraphOpt *opt, float v) {
  // Check arguments
  if (opt == NULL || v < 0.0 || v > 1.0)
    return;
  // Set the curvature
  opt->_curvature = v;
}

// Set the mode of display for nodes' label to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetNodeLabelMode(CloudGraphOpt *opt,
  CloudGraphOptNodeLabel mode) {
  // Check arguments
  if (opt == NULL)
    return;
  // Set the mode
  opt->_nodeLabelMode = mode;
}

// Set the mode of display for families' label to 'mode'
// Do nothing if arguments are invalid
void CloudGraphOptSetFamilyLabelMode(CloudGraphOpt *opt,
  CloudGraphOptFamilyLabel mode) {
  // Check arguments
  if (opt == NULL)
    return;
  // Set the mode
  opt->_familyLabelMode = mode;
}

// Set the font size for nodes' label to 'size'
// Do nothing if arguments are invalid
void CloudGraphOptSetFontSizeNode(CloudGraphOpt *opt, float size) {
  // Check arguments
  if (opt == NULL || size <= 0.0)
    return;
  // Set the size
  opt->_fontSizeNode = size;
}

// Set the font size for families' label to 'size'
// Do nothing if arguments are invalid
void CloudGraphOptSetFontSizeFamily(CloudGraphOpt *opt, float size) {
  // Check arguments
  if (opt == NULL || size <= 0.0)
    return;
  // Set the size
  opt->_fontSizeFamily = size;
```

```
}

// Return the length of the longest displayed node label
// Return 0.0 of arguments are invalid or there is no displayed label
float CloudGraphGetMaxLengthLblNode(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return 0.0;
  // Declare a variable to memorize the size of the longest label
  float maxLength = 0.0;
  // If the label of nodes is displayed
  if (opt->_nodeLabelMode != CloudGraphOptNodeLabelNone) {
    // Declare a pointer to go through the sets
    GSetElem *ptr = cloud->_nodes->_head;
    // Loop through nodes
    while (ptr != NULL) {
      CloudGraphNode *node = (CloudGraphNode*)(ptr->_data);
      // If the bounding box is larger than the current max length
      float l = VecGet(node->_boundingBox->_axis[0], 0);
      if (l > maxLength)
        // Update the max length;
        maxLength = l;
      // Move to the next node
      ptr = ptr->_next;
    }
  }
  // Return the result
  return maxLength;
}

// Return the length of the longest displayed family label
// Return 0.0 of arguments are invalid or there is no displayed label
float CloudGraphGetMaxLengthLblFamily(CloudGraph *cloud, CloudGraphOpt *opt) {
  // Check arguments
  if (cloud == NULL || opt == NULL)
    return 0.0;
  // Declare a variable to memorize the size of the longest label
  float maxLength = 0.0;
  // If the label of nodes is displayed
  if (opt->_familyLabelMode != CloudGraphOptFamilyLabelNone) {
    // Declare a pointer to go through the sets
    GSetElem *ptr = cloud->_families->_head;
    // Loop through families
    while (ptr != NULL) {
      CloudGraphFamily *family = (CloudGraphFamily*)(ptr->_data);
      // If the bounding box is larger than the current max length
      float l = VecGet(family->_boundingBox->_axis[0], 0);
      if (l > maxLength)
        // Update the max length;
        maxLength = l;
      // Move to the next family
      ptr = ptr->_next;
    }
  }
  // Return the result
  return maxLength;
}

// Return the node 'id' or NULL if arguments are invalid
CloudGraphNode* CloudGraphGetNode(CloudGraph *cloud, int id) {
  // Check arguments
  if (cloud == NULL)
```

```
    return NULL;
  // Declare a pointer for the result
  CloudGraphNode *res = NULL;
  // Declare a pointer on the set of node
  GSetElem *elem = cloud->_nodes->_head;
  // Loop on node, stop when the node is found
  while (elem != NULL && res == NULL) {
    // If the current node is the searched node
    if (((CloudGraphNode*)(elem->_data))->_id == id)
      // Update the result
      res = elem->_data;
    // Move to next node
    elem = elem->_next;
  }
  // Return the result
  return res;
}

// Return the family 'id' or NULL if arguments are invalid
CloudGraphFamily* CloudGraphGetFamily(CloudGraph *cloud, int id) {
  // Check arguments
  if (cloud == NULL)
    return NULL;
  // Declare a pointer for the result
  CloudGraphFamily *res = NULL;
  // Declare a pointer on the set of family
  GSetElem *elem = cloud->_families->_head;
  // Loop on node, stop when the family is found
  while (elem != NULL && res == NULL) {
    // If the current family is the searched family
    if (((CloudGraphFamily*)(elem->_data))->_id == id)
      // Update the result
      res = elem->_data;
    // Move to next family
    elem = elem->_next;
  }
  // Return the result
  return res;
}
```

4 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_DEBUG)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

main: main.o cloudgraph.o Makefile $(LIBPATH)/tgapaint.o $(LIBPATH)/gset.o $(LIBPATH)/bcurve.o $(LIBPATH)/pbmath.o
gcc  $(OPTIONS) main.o $(LIBPATH)/tgapaint.o $(LIBPATH)/gset.o $(LIBPATH)/bcurve.o $(LIBPATH)/pbmath.o cloudgraph.o -

main.o : main.c cloudgraph.h Makefile
```

```
gcc $(OPTIONS) -I$(INCPATH) -c main.c

cloudgraph.o : cloudgraph.c cloudgraph.h $(INCPATH)/tgapaint.h $(INCPATH)/gset.h $(INCPATH)/pbmath.h $(INCPATH)/bcurv
gcc $(OPTIONS) -I$(INCPATH) -c cloudgraph.c

clean :
rm -rf *.o main

test :
main -file testCloud.txt -tga cloud.tga -line -nodeLabel -familyLabel

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main -tga cloud.tga
```

5 Usage

```
// ============ MAIN.C ================

// ================= Include ==================

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "cloudgraph.h"

// ================= Main function ==================

int main(int argc, char **argv) {
  // Initialize the random generator
  time_t seed = time(NULL);
  srandom(seed);
  // Create the CloudGraph
  CloudGraph *cloud = CloudGraphCreate();
  // If we couldn't create the CloudGraph
  if (cloud == NULL) {
    // Display a message and stop
    fprintf(stderr, "Couldn't create the CloudGraph\n");
    return 1;
  }
  // Declare variables to memorize the arguments and set default values
  char flagPrint = 0;
  char *fileNameTGA = NULL;
  char *fileNameGraph = NULL;
  int nbNodeMin = 5;
  int nbNodeMax = 20;
  int nbFamilyMin = 1;
  int nbFamilyMax = 5;
  float density = 0.1;
  CloudGraphMode mode = CloudGraphModeLine;
  // Declare a variable for the graphical options when exporting to TGA
  CloudGraphOpt *opt = CloudGraphOptCreate();
  // If we couldn't create the CloudGraphOpt
  if (opt == NULL) {
    // Display a message
    fprintf(stderr, "Couldn't create the CloudGraphOpt\n");
    // Free memory
    CloudGraphFree(&cloud);
```

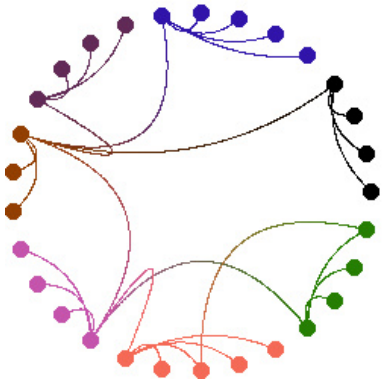
```
  // Stop here
  return 1;
}
// Decode arguments
for (int iArg = 0; iArg < argc; ++iArg) {
  if (strcmp(argv[iArg] , "-tga") == 0 && iArg + 1 < argc) {
    fileNameTGA = argv[iArg + 1];
    ++iArg;
  } else if (strcmp(argv[iArg] , "-print") == 0) {
    flagPrint = 1;
  } else if (strcmp(argv[iArg] , "-curved") == 0 && iArg + 1 < argc) {
    CloudGraphOptSetCurvedLink(opt, true);
    float curvature = atof(argv[iArg + 1]);
    CloudGraphOptSetCurvature(opt, curvature);
    ++iArg;
  } else if (strcmp(argv[iArg] , "-circle") == 0) {
    mode = CloudGraphModeCircle;
  } else if (strcmp(argv[iArg] , "-line") == 0) {
    mode = CloudGraphModeLine;
  } else if (strcmp(argv[iArg] , "-nodeLabel") == 0) {
    CloudGraphOptSetNodeLabelMode(opt, CloudGraphOptNodeLabelAll);
  } else if (strcmp(argv[iArg] , "-familyLabel") == 0) {
    CloudGraphOptSetFamilyLabelMode(opt, CloudGraphOptFamilyLabelAll);
  } else if (strcmp(argv[iArg] , "-file") == 0 && iArg + 1 < argc) {
    fileNameGraph = argv[iArg + 1];
    ++iArg;
  } else if (strcmp(argv[iArg] , "-rnd") == 0 && iArg + 5 < argc) {
    nbNodeMin = atoi(argv[iArg + 1]);
    nbNodeMax = atoi(argv[iArg + 2]);
    nbFamilyMin = atoi(argv[iArg + 3]);
    nbFamilyMax = atoi(argv[iArg + 4]);
    density = atof(argv[iArg + 5]);
    iArg += 5;
  } else if (strcmp(argv[iArg] , "-help") == 0) {
    printf("arguments : [-tga <filename>] [-print]");
    printf(" [-file <filename>] [-free] [-circle] [-line]");
    printf(" [-rnd <nbNodeMin> <nbNodeMax> <nbFamilyMin>");
    printf(" <nbFamilyMax> <density>]");
    printf(" <-nodeLabel> <-familyLabel>");
    printf(" [-curved <curvature in [0.0,1.0]>]\n");
    printf("if -rnd and -file are both omitted, uses ");
    printf("'-rnd %d %d %d %d %f' by default\n", nbNodeMin, nbNodeMax,
      nbFamilyMin, nbFamilyMax, density);
    // Stop here
    CloudGraphFree(&cloud);
    CloudGraphOptFree(&opt);
    return 0;
  }
}
// Set the mode
CloudGraphOptSetMode(opt, mode);
// If there is no input file
if (fileNameGraph == NULL) {
  // Create a random graph
  bool ret = CloudGraphCreateRnd(&cloud, nbNodeMin,
    nbNodeMax, nbFamilyMin, nbFamilyMax, density);
  // If we couldn't initialize the CloudGraph
  if (ret != true) {
    // Display a message
    fprintf(stderr,
      "Error while creating the random graph\n");
    // Free the memory
```

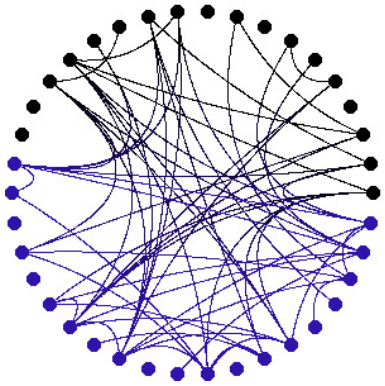
```
    CloudGraphFree(&cloud);
    CloudGraphOptFree(&opt);
    // Stop here
    return 1;
  }
// Else there is a input file
} else {
  // Load the input file
  FILE *stream = fopen(fileNameGraph, "r");
  int ret = CloudGraphLoad(&cloud, stream);
  // If we couldn't load the CloudGraph
  if (ret != 0) {
    // Display a message
    fprintf(stderr,
      "Error while loading the CloudGraph file (%d)\n", ret);
    // Free the memory
    CloudGraphFree(&cloud);
    CloudGraphOptFree(&opt);
    // Stop here
    return 1;
  }
  fclose(stream);
}
// Arrange the CloudGraph
bool ret = CloudGraphArrange(cloud, opt);
if (ret == false) {
  // Display a message
  fprintf(stderr, "Error while arranging the nodes\n");
  // Free the memory
  CloudGraphFree(&cloud);
  CloudGraphOptFree(&opt);
  // Stop here
  return 1;
}
// If there is a output TGA file
if (fileNameTGA != NULL) {
  // Save the result in the TGA picture
  TGA *tga = CloudGraphToTGA(cloud, opt);
  if (tga == NULL) {
    // Display a message
    fprintf(stderr, "Error while exporting to TGA\n");
    // Free the memory
    CloudGraphFree(&cloud);
    CloudGraphOptFree(&opt);
    // Stop here
    return 1;
  }
  int ret = TGASave(tga, fileNameTGA);
  if (ret != 0) {
    // Display a message
    fprintf(stderr, "Error while saving TGA\n");
    // Free the memory
    CloudGraphFree(&cloud);
    CloudGraphOptFree(&opt);
    // Stop here
    return 1;
  }
  // Free the memory used by the TGA
  TGAFree(&tga);
}
// If the user requested printing of the CloudGraph
if (flagPrint == 1) {
```

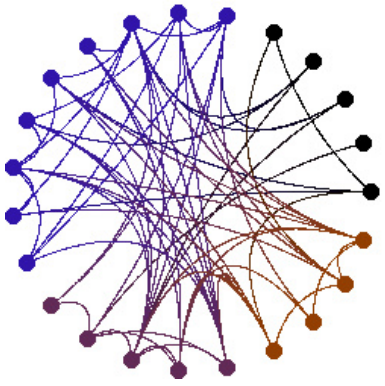
```
    // Print the cloud
    CloudGraphPrint(cloud, stdout);
  }
  // Free memory
  CloudGraphFree(&cloud);
  CloudGraphOptFree(&opt);
  // Return the success code
  return 0;
}
```

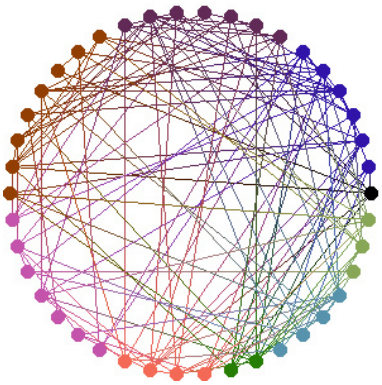
Output:

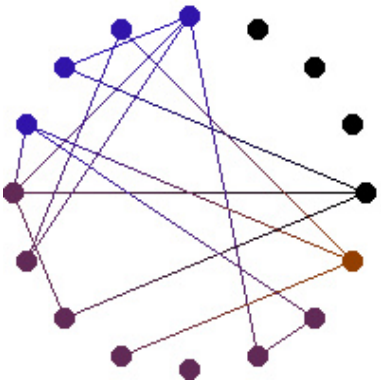
```
main -print

Families:
#0 rgb(249,012,211) Family000
#1 rgb(110,015,053) Family001
Nodes:
#0 family(0) <0.000,18.000> Node000
#1 family(0) <0.000,54.000> Node001
#3 family(0) <0.000,90.000> Node003
#4 family(0) <0.000,126.000> Node004
#6 family(0) <0.000,162.000> Node006
#8 family(0) <0.000,198.000> Node008
#11 family(0) <0.000,234.000> Node011
#13 family(0) <0.000,270.000> Node013
#15 family(0) <0.000,306.000> Node015
#2 family(1) <0.000,342.000> Node002
#5 family(1) <0.000,378.000> Node005
#7 family(1) <0.000,414.000> Node007
#9 family(1) <0.000,450.000> Node009
#10 family(1) <0.000,486.000> Node010
#12 family(1) <0.000,522.000> Node012
#14 family(1) <0.000,558.000> Node014
Links:
000-006
001-007
002-013
003-005
004-010
005-010
007-013
009-015
011-012
```



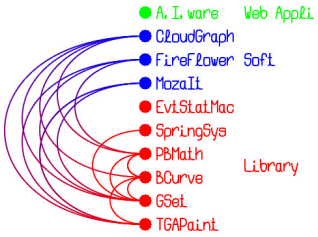









```
main -file testCloud.txt -nodeLabel -familyLabel
```



testCloud.txt :

```
3
0 255 0 0 Library
1 0 0 255 Soft
2 0 255 0 Web Appli
10
0 0 TGAPaint
1 0 GSet
2 1 MozaIt
3 1 FireFlower
4 0 BCurve
5 0 PBMath
6 0 SpringSys
7 0 EvtStatMac
8 2 A.I.ware
9 1 CloudGraph
16
4 5
4 1
5 1
5 4
6 1
0 4
9 1
9 5
9 4
9 0
3 1
3 5
3 0
2 1
2 0
```