CloudGraph

P. Baillehache

June 16, 2017

Contents

1 Position of nodes **2**
- 1.1 Circle . . . 2
- 1.2 Spring-mass system . . . 3

2 Interface **3**

3 Code **5**

4 Makefile **14**

5 Usage **15**

6 Examples **19**
- 6.1 testCloud1.txt . . . 19
- 6.2 testCloud2.txt . . . 20
- 6.3 testCloud3.txt . . . 20
- 6.4 testCloud4.txt . . . 21
- 6.5 Random graphs . . . 25

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. It uses the TGA library and SpringSys library functions.

The representation of the graph has 2 modes: circular and free. The representation of the links has 2 modes: straight line and curved line. Categories are represented by different color (default setup and manual setup possible), and links between two categories have shading colors.

1 Position of nodes

The CloudGraph's algorithm calculates the position of nodes such as nodes of a same category or linked in the network are nearer from each other than from nodes unlinked or from another category.

There are two modes of arrangement : the circle mode where points are placed on the perimeter of a circle, and the spring-mass system mode where nodes and edges are converted into a spring-mass system whose equilibrium state gives the nodes' position.

1.1 Circle

In circle mode the calculation of nodes' coordinates is made as follow:

1. Nodes are placed, ordered by category, on the circumference of a circle of perimeter equals to the number of nodes.
2. Positions are normalized to fit the cloud into the unit square.

Links between nodes, if curved, are spline whose control point is: the center of the circle for a link between nodes of different categories; $\overrightarrow{Ctrl} = 0.75\overrightarrow{Af} + 0.25\overrightarrow{C}$ where $\overrightarrow{C}$ is the center of the circle and $\overrightarrow{Af}$ is the average position of the family's nodes.

1.2 Spring-mass system

In spring-mass system mode the calculation of nodes' coordinates is made as follow:

1. Nodes are placed, ordered by category, on the circumference of a circle of perimeter equals to the number of nodes.
2. A spring-mass system is created, where masses are nodes of the graph and springs are edges of the graph.
3. For each node springs are added toward its two nearest neighbours in the same category (if they are not already connected in the graph).
4. For each family springs are added between its leader and the two nearest family's leader (if there is not already such a link), where the leader of a family is defined as the node in this family having the most link in the graph for this family (eventually chosen randomly if there are several nodes with the same number of link).
5. Spring's rest distance is set to 1.0 for spring connecting two masses of same category, and to 4.0 else.
6. The spring-mass system is solved.
7. The result is normalized to fit into the unit square.

Links between nodes, if curved, are spline whose control point is: the center of the family if both nodes are in the same family; the average of the center of the family of each node.

2 Interface

```
// ============ CloudGraph.h ================

#ifndef __CLOUDGRAPH_H
#define __CLOUDGRAPH_H
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "/home/bayashi/Coding/SpringSys/C/SpringSys.h"

#define CG_NBMAXFAMILY 100
```

```
typedef enum {
  CloudGraphModeCircle, CloudGraphModeSpring
} CloudGraphMode;

typedef struct {
  float _pos[2];
  float _posTmp[2];
  int _id;
  int _family;
  int _nbLink;
  int _index;
} CloudGraphNode;

typedef struct {
  int _nodes[2];
} CloudGraphEdge;

typedef struct {
  int _nbNode;
  int _nbEdge;
  int _nbFamily;
  CloudGraphNode *_nodes;
  CloudGraphEdge *_edges;
  unsigned char _colors[3 * CG_NBMAXFAMILY];
  char _familyLink;
  CloudGraphMode _mode;
  char _curvedLink;
} CloudGraph;

// Create a new CloudGraph
CloudGraph* CloudGraphCreate();

// Free the memory used by a CloudGraph
void CloudGraphFree(CloudGraph *cloud);

// Load the graph from a text file
// Return 0 on success, else
// 1: invalid arguments
// 2: memory allocation failed
// 3: invalid number of node or edge
// 4: premature end of file
// 5: can't open the file
// The format of the text file is as follow:
// 1st line : <nb node> <nb edge>
// <nb node> following lines : <node id (int)> <node family (int)>
// <nb edge> following lines : <1st node id> <2nd node id>
int CloudGraphLoad(CloudGraph *cloud, char *fileName);

// Set the graph from data in memory
// Return 0 on success, else
// 1: invalid arguments
// 2: memory allocation failed
// 3: invalid number of node or edge
// The format of the data is as follow:
// nodes : array of (<node id>, <node family>)
// edges : array of (<1st node index>, <2nd node index>)
int CloudGraphSet(CloudGraph *cloud, int nbNode, int nbEdge,
  int *nodes, int *edges);

// Calculate the position of the nodes according to the edge
void CloudGraphArrange(CloudGraph *cloud);
```

```
// Create a random graph for test purpose
// srandom() have been called prior to calling this function
// between nbNodeMin and nbNodeMax nodes,
// nbFamily categories
// connectivity rate between 0.01 and 0.05
// Return 0 on success, else
// 1: invalid arguments
// 2: memory allocation failed
int CloudGraphCreateRnd(CloudGraph *cloud, int nbNodeMin,
  int nbNodeMax, int nbFamily);

// Print the node position on the given stream
// one node per line, format as: id x y
// coordinate are contains in the unit square [(0.0,0.0),(1.0,1.0)]
// don't do anything if the arguments are invalid
void CloudGraphPrint(CloudGraph *cloud, FILE* stream);

// Get the stretch of the cloud
// 0.0 means the nodes are perfectly arranged,
// else its the sum of distance between actual position and
// ideal position
// return -1.0 if the argument is invalid
float CloudGraphStretch(CloudGraph *cloud);

// Get the node identified by id
// return NULL if invalid arguments
CloudGraphNode* CloudGraphGetNodeById(CloudGraph *cloud, int id);

// Sort the nodes by family
// do nothing if invalid arguments
void CloudGraphSortNodeByFamily(CloudGraph *cloud);

// Set the color for the family
// do nothing if invalid arguments
void CloudGraphSetFamilyColor(CloudGraph *cloud, int family,
  unsigned char *rgb);

// Get the average position  pos of nodes of family iFamily
// (0.0, 0.0) if invalid arguments or the family has no node
void CloudGraphGetFamilyCenter(CloudGraph *cloud, int iFamily,
  float *pos);

#endif
```

3 Code

```
// ============ CloudGraph.c ================

#include "CloudGraph.h"

#define CG_DISTCOUSIN 4.0
#define CG_DISTBROTHER 1.0

CloudGraph* CloudGraphCreate() {
  CloudGraph *cloud = (CloudGraph*)malloc(sizeof(CloudGraph));
  cloud->_nbNode = 0;
  cloud->_nbEdge = 0;
  cloud->_nbFamily = 0;
  cloud->_nodes = NULL;
  cloud->_edges = NULL;
```

```
  cloud->_familyLink = 1;
  cloud->_curvedLink = 0;
  cloud->_mode = CloudGraphModeCircle;
  cloud->_colors[0] = cloud->_colors[1] = cloud->_colors[2] = 0;
  for (int iCol = 1; iCol < CG_NBMAXFAMILY; ++iCol) {
    cloud->_colors[3 * iCol] =
      (cloud->_colors[3 * (iCol - 1)] + 49) % 256;
    cloud->_colors[3 * iCol + 1] =
      (cloud->_colors[3 * (iCol - 1) + 1] + 21) % 256;
    cloud->_colors[3 * iCol + 2] =
      (cloud->_colors[3 * (iCol - 1) + 2] + 171) % 256;
  }
  return cloud;
}

void CloudGraphFree(CloudGraph *cloud) {
  if (cloud == NULL) return;
  if (cloud->_nodes != NULL) free(cloud->_nodes);
  if (cloud->_edges != NULL) free(cloud->_edges);
  cloud->_nbNode = 0;
  cloud->_nbEdge = 0;
}

CloudGraphNode* CloudGraphGetNodeById(CloudGraph *cloud, int id) {
  if (cloud == NULL) return NULL;
  CloudGraphNode *ret = NULL;
  for (int iNode = 0; iNode < cloud->_nbNode && ret == NULL; ++iNode) {
    if (cloud->_nodes[iNode]._id == id)
      ret = &(cloud->_nodes[iNode]);
  }
  return ret;
}

int CloudGraphGetNodeIndexById(CloudGraph *cloud, int id) {
  if (cloud == NULL) return 0;
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    if (cloud->_nodes[iNode]._id == id)
      return iNode;
  }
  return 0;
}

int CloudGraphLoad(CloudGraph *cloud, char *fileName) {
  if (cloud == NULL || fileName == NULL) return 1;
  CloudGraphFree(cloud);
  FILE *f = fopen(fileName, "r");
  if (f == NULL) return 5;
  fscanf(f, "%d %d\n", &(cloud->_nbNode), &(cloud->_nbEdge));
  if (cloud->_nbNode <= 0 || cloud->_nbEdge < 0) {
    fclose(f);
    return 3;
  }
  cloud->_nodes =
    (CloudGraphNode*)malloc(sizeof(CloudGraphNode) * cloud->_nbNode);
  if (cloud->_nodes == NULL) {
    fclose(f);
    return 2;
  }
  cloud->_edges =
    (CloudGraphEdge*)malloc(sizeof(CloudGraphEdge) * cloud->_nbEdge);
  if (cloud->_edges == NULL) {
    fclose(f);
```

```
    return 2;
  }
  int iNode, iEdge;
  for (iNode = 0; iNode < cloud->_nbNode && !feof(f); ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    fscanf(f, "%d %d\n", &(node->_id), &(node->_family));
    if (node->_family >= cloud->_nbFamily)
      cloud->_nbFamily = node->_family + 1;
    node->_nbLink = 0;
    node->_index = iNode;
  }
  for (iEdge = 0; iEdge < cloud->_nbEdge && !feof(f); ++iEdge) {
    CloudGraphEdge *edge = &(cloud->_edges[iEdge]);
    fscanf(f, "%d %d\n", &(edge->_nodes[0]), &(edge->_nodes[1]));
    CloudGraphNode *nodeA =
      CloudGraphGetNodeById(cloud, edge->_nodes[0]);
    ++(nodeA->_nbLink);
    CloudGraphNode *nodeB =
      CloudGraphGetNodeById(cloud, edge->_nodes[1]);
    ++(nodeB->_nbLink);
  }
  if (iNode != cloud->_nbNode || iEdge != cloud->_nbEdge) {
    fclose(f);
    return 4;
  }
  fclose(f);
  return 0;
}

int CloudGraphSet(CloudGraph *cloud, int nbNode, int nbEdge,
  int *nodes, int *edges) {
  if (cloud == NULL || nodes == NULL || edges == NULL ||
    nbNode <= 0 || nbEdge < 0) return 1;
  CloudGraphFree(cloud);
  cloud->_nbNode = nbNode;
  cloud->_nbEdge = nbEdge;
  cloud->_nodes =
    (CloudGraphNode*)malloc(sizeof(CloudGraphNode) * cloud->_nbNode);
  if (cloud->_nodes == NULL) {
    return 2;
  }
  cloud->_edges =
    (CloudGraphEdge*)malloc(sizeof(CloudGraphEdge) * cloud->_nbEdge);
  if (cloud->_edges == NULL) {
    return 2;
  }
  int iNode, iEdge;
  for (iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    node->_id = nodes[2 * iNode];
    node->_family = nodes[2 * iNode + 1];
    node->_nbLink = 0;
  }
  for (iEdge = 0; iEdge < cloud->_nbEdge; ++iEdge) {
    CloudGraphEdge *edge = &(cloud->_edges[iEdge]);
    edge->_nodes[0] = edges[2 * iEdge];
    edge->_nodes[1] = edges[2 * iEdge + 1];
    CloudGraphNode *nodeA =
      CloudGraphGetNodeById(cloud, edge->_nodes[0]);
    ++(nodeA->_nbLink);
    CloudGraphNode *nodeB =
      CloudGraphGetNodeById(cloud, edge->_nodes[1]);
```

```
    ++(nodeB->_nbLink);
  }
  return 0;
}

// Return the two nearest brothers in the same category
// result in brother[2], -1 if no brother
void CloudGraphGetNearestBrothers(CloudGraph *cloud,
  CloudGraphNode *node, int iNode, int *brother) {
  brother[0] = brother[1] = -1;
  float dBrother[2] = {0.0, 0.0};
  for (int jNode = 0; jNode < cloud->_nbNode; ++jNode) {
    CloudGraphNode *nodeA = &(cloud->_nodes[jNode]);
    if (jNode != iNode && nodeA->_family == node->_family) {
      float d = sqrt(pow(node->_pos[0] - nodeA->_pos[0], 2.0) +
        pow(node->_pos[1] - nodeA->_pos[1], 2.0));
      if (brother[0] == -1) {
        brother[0] = jNode;
        dBrother[0] = d;
      } else if (d < dBrother[0]) {
        brother[1] = brother[0];
        dBrother[1] = dBrother[0];
        brother[0] = jNode;
        dBrother[0] = d;
      } else if (brother[1] == -1) {
        brother[1] = jNode;
        dBrother[1] = d;
      } else if (d < dBrother[1]) {
        brother[1] = jNode;
        dBrother[1] = d;
      }
    }
  }
}

CloudGraphNode* CloudGraphGetFamilyLeader(CloudGraph *cloud,
  int iFamily) {
  CloudGraphNode *ret = NULL;
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    if (node->_family == iFamily) {
      if (ret == NULL) {
        ret = node;
      } else {
        if (node->_nbLink > ret->_nbLink)
          ret = node;
      }
    }
  }
  return ret;
}

// Return the two nearest leaders in different family
// result in leader[2], -1 if no leader
void CloudGraphGetNearestLeaders(CloudGraph *cloud,
  CloudGraphNode *nodeLeader, int iFamily, int *leader) {
  leader[0] = leader[1] = -1;
  float dLeader[2] = {0.0, 0.0};
  for (int jFamily = 0; jFamily < cloud->_nbFamily; ++jFamily) {
    CloudGraphNode *nodeA =
      CloudGraphGetFamilyLeader(cloud, jFamily);
    if (jFamily != iFamily && nodeA != NULL) {
```

```
      float d = sqrt(pow(nodeLeader->_pos[0] - nodeA->_pos[0], 2.0) +
        pow(nodeLeader->_pos[1] - nodeA->_pos[1], 2.0));
      if (leader[0] == -1) {
        leader[0] = nodeA->_index;
        dLeader[0] = d;
      } else if (d < dLeader[0]) {
        leader[1] = leader[0];
        dLeader[1] = dLeader[0];
        leader[0] = nodeA->_index;
        dLeader[0] = d;
      } else if (leader[1] == -1) {
        leader[1] = nodeA->_index;
        dLeader[1] = d;
      } else if (d < dLeader[1]) {
        leader[1] = nodeA->_index;
        dLeader[1] = d;
      }
    }
  }
}

float CloudGraphStretch(CloudGraph *cloud) {
  if (cloud == NULL) return -1.0;
  float ret = 0.0;
  for (int iNode = 0; iNode < cloud->_nbNode - 1; ++iNode) {
    CloudGraphNode *nodeA = &(cloud->_nodes[iNode]);
    int brother[2];
    CloudGraphGetNearestBrothers(cloud, nodeA, iNode, brother);
    for (int iEdge = 0; iEdge < cloud->_nbEdge; ++iEdge) {
      CloudGraphEdge *edge = &(cloud->_edges[iEdge]);
      if ((edge->_nodes[0] == iNode && edge->_nodes[1] == brother[0]) ||
        (edge->_nodes[1] == iNode && edge->_nodes[0] == brother[0]))
        brother[0] = -1;
      if ((edge->_nodes[0] == iNode && edge->_nodes[1] == brother[1]) ||
        (edge->_nodes[1] == iNode && edge->_nodes[0] == brother[1]))
        brother[1] = -1;
    }
    if (brother[0] != -1) {
      CloudGraphNode *nodeB = &(cloud->_nodes[brother[0]]);
      float D = sqrt(pow(nodeA->_pos[0] - nodeB->_pos[0], 2.0) +
        pow(nodeA->_pos[1] - nodeB->_pos[1], 2.0));
      ret += fabs(D - CG_DISTBROTHER);
    }
    if (brother[1] != -1) {
      CloudGraphNode *nodeB = &(cloud->_nodes[brother[1]]);
      float D = sqrt(pow(nodeA->_pos[0] - nodeB->_pos[0], 2.0) +
        pow(nodeA->_pos[1] - nodeB->_pos[1], 2.0));
      ret += fabs(D - CG_DISTBROTHER);
    }
  }
  for (int iEdge = 0; iEdge < cloud->_nbEdge; ++iEdge) {
    CloudGraphEdge *edge = &(cloud->_edges[iEdge]);
    CloudGraphNode *nodeA =
      CloudGraphGetNodeById(cloud, edge->_nodes[0]);
    CloudGraphNode *nodeB =
      CloudGraphGetNodeById(cloud, edge->_nodes[1]);
    float D = sqrt(pow(nodeA->_pos[0] - nodeB->_pos[0], 2.0) +
      pow(nodeA->_pos[1] - nodeB->_pos[1], 2.0));
    if (nodeA->_family == nodeB->_family) {
      ret += fabs(D - CG_DISTBROTHER);
    } else {
      ret += fabs(D - CG_DISTCOUSIN);
```

```
    }
  }
  return ret;
}

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

void CloudGraphSortNodeByFamily(CloudGraph *cloud) {
  if (cloud == NULL) return;
  for (int iNode = 0; iNode < cloud->_nbNode - 1; ++iNode) {
    for (int jNode = iNode + 1; jNode < cloud->_nbNode; ++jNode) {
      CloudGraphNode *nodeA = &(cloud->_nodes[iNode]);
      CloudGraphNode *nodeB = &(cloud->_nodes[jNode]);
      if (nodeA->_family > nodeB->_family) {
        CloudGraphNode tmp;
        memcpy(&tmp, nodeA, sizeof(CloudGraphNode));
        memcpy(nodeA, nodeB, sizeof(CloudGraphNode));
        memcpy(nodeB, &tmp, sizeof(CloudGraphNode));
        nodeA->_index = iNode;
        nodeB->_index = jNode;
      }
    }
  }
}

void CloudGraphInitNodePos(CloudGraph *cloud) {
  // Arrange nodes equally spaced along a circle of radius such as
  // perimeter is equal to number of nodes
  float p = (float)(cloud->_nbNode);
  float twoPi = 2.0 * 3.14159;
  float r = p / twoPi;
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    float a = (float)iNode / (float)(cloud->_nbNode) * twoPi;
    node->_pos[0] = r * (cos(a) + 1.0);
    node->_pos[1] = r * (sin(a) + 1.0);
  }
}

void CloudGraphNormalize(CloudGraph *cloud) {
  float range[4];
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    if (iNode == 0) {
      range[0] = node->_pos[0]; range[1] = node->_pos[1];
      range[2] = node->_pos[0]; range[3] = node->_pos[1];
    } else {
      if (range[0] > node->_pos[0]) range[0] = node->_pos[0];
      if (range[1] > node->_pos[1]) range[1] = node->_pos[1];
      if (range[2] < node->_pos[0]) range[2] = node->_pos[0];
      if (range[3] < node->_pos[1]) range[3] = node->_pos[1];
    }
  }
  float dim[2];
  dim[0] = range[2] - range[0]; dim[1] = range[3] - range[1];
  float trans[2];
  trans[0] = -1.0 * range[0]; trans[1] = -1.0 * range[1];
  float scale[2];
  scale[0] = (dim[0] > 0.0001 ? 1.0 / dim[0] : 1.0);
  scale[1] = (dim[1] > 0.0001 ? 1.0 / dim[1] : 1.0);
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
```

```
    node->_pos[0] = (node->_pos[0] + trans[0]) * scale[0];
    node->_pos[1] = (node->_pos[1] + trans[1]) * scale[1];
  }
}

char CloudGraphSpringExist(int *spring, int nbSpring, int iMass, int jMass) {
  char flag = 0;
  for (int iSpring = 0; iSpring < nbSpring && flag == 0;
    ++iSpring) {
    if ((spring[iSpring * 2] == iMass &&
      spring[iSpring * 2 + 1] == jMass) ||
      (spring[iSpring * 2 + 1] == iMass &&
      spring[iSpring * 2] == jMass)) {
      flag = 1;
    }
  }
  return flag;
}

void CloudGraphArrange(CloudGraph *cloud) {
  if (cloud == NULL) return;
  // Sort the nodes by family
  CloudGraphSortNodeByFamily(cloud);
  // Initialise arrangement of nodes
  CloudGraphInitNodePos(cloud);
  if (cloud->_mode == CloudGraphModeSpring) {
    for (int iStep = 0; iStep < 2; ++iStep) {
      // Create the equivalent spring-mass system and solve it
      SpringSys *sys = SpringSysCreate();
      int nbMass = cloud->_nbNode;
      float *mass = (float*)malloc(sizeof(float) * nbMass * 2);
      if (mass == NULL) {
        return;
      }
      char *fixed = (char*)malloc(sizeof(char) * nbMass);
      if (fixed == NULL) {
        free(mass);
        return;
      }
      for (int iMass = 0; iMass < nbMass; ++iMass) {
        CloudGraphNode *node = &(cloud->_nodes[iMass]);
        fixed[iMass] = 0;
        mass[iMass * 2] = node->_pos[0];
        mass[iMass * 2 + 1] = node->_pos[1];
      }
      int nbSpring = cloud->_nbEdge + 2 * cloud->_nbNode +
        cloud->_nbFamily;
      int *spring = (int*)malloc(sizeof(int) * nbSpring * 2);
      if (spring == NULL) {
        free(mass);
        free(fixed);
        return;
      }
      float *coeff = (float*)malloc(sizeof(float) * nbSpring * 2);
      if (coeff == NULL) {
        free(mass);
        free(fixed);
        free(spring);
        return;
      }
      for (int iSpring = 0; iSpring < cloud->_nbEdge; ++iSpring) {
        CloudGraphEdge *edge = &(cloud->_edges[iSpring]);
```

```
    spring[iSpring * 2] =
      CloudGraphGetNodeIndexById(cloud, edge->_nodes[0]);
    spring[iSpring * 2 + 1] =
      CloudGraphGetNodeIndexById(cloud, edge->_nodes[1]);
    coeff[iSpring * 2] = 1.0;
    if (CloudGraphGetNodeById(cloud, edge->_nodes[0])->_family ==
      CloudGraphGetNodeById(cloud, edge->_nodes[1])->_family)
      coeff[iSpring * 2 + 1] = CG_DISTBROTHER;
    else
      coeff[iSpring * 2 + 1] = CG_DISTCOUSIN;
  }
  nbSpring = cloud->_nbEdge;
  if (cloud->_familyLink == 1) {
    // add springs between brothers to keep family grouped
    for (int iMass = 0; iMass < nbMass; ++iMass) {
      CloudGraphNode *node = &(cloud->_nodes[iMass]);
      int brother[2];
      CloudGraphGetNearestBrothers(cloud, node, iMass, brother);
      for (int iBrother = 0;
        iBrother < 2 && brother[iBrother] != -1; ++iBrother) {
        char flag = CloudGraphSpringExist(spring, nbSpring, iMass, brother[iBrother]);
        if (flag == 0) {
          spring[nbSpring * 2] = iMass;
          spring[nbSpring * 2 + 1] = brother[iBrother];
          coeff[nbSpring * 2] = 1.0;
          coeff[nbSpring * 2 + 1] = CG_DISTBROTHER;
          ++nbSpring;
        }
      }
    }
    // add springs between families to keep families away from
    // each others
    for (int iFamily = 0; iFamily < cloud->_nbFamily; ++iFamily) {
      CloudGraphNode *nodeLeader =
        CloudGraphGetFamilyLeader(cloud, iFamily);
      if (nodeLeader != NULL) {
        int leader[2];
        CloudGraphGetNearestLeaders(cloud, nodeLeader,
          iFamily, leader);
        for (int iLeader = 0;
          iLeader < 2 && leader[iLeader] != -1; ++iLeader) {
          char flag = CloudGraphSpringExist(spring, nbSpring, nodeLeader->_index, leader[iLea
          if (flag == 0) {
            spring[nbSpring * 2] = nodeLeader->_index;
            spring[nbSpring * 2 + 1] = leader[iLeader];
            coeff[nbSpring * 2] = 1.0;
            coeff[nbSpring * 2 + 1] = CG_DISTCOUSIN;
            ++nbSpring;
          }
        }
      }
    }
  }
  SpringSysSet(sys, nbMass, nbSpring, mass, fixed, spring, coeff);
  SpringSysFindEquilibrium(sys);
  // Update the nodes' position
  for (int iMass = 0; iMass < sys->_nbMass; ++iMass) {
    SpringSysMass *mass = &(sys->_masses[iMass]);
    CloudGraphNode *node = &(cloud->_nodes[iMass]);
    node->_pos[0] = mass->_pos[0];
    node->_pos[1] = mass->_pos[1];
  }
```

```
      free(mass);
      free(fixed);
      free(spring);
      free(coeff);
    }
  }
  // Scale and translate nodes to fit the unit square
  CloudGraphNormalize(cloud);
}

int CloudGraphCreateRnd(CloudGraph *cloud, int nbNodeMin,
  int nbNodeMax, int nbFamily) {
  if (cloud == NULL || nbNodeMin > nbNodeMax || nbNodeMin < 0 ||
    nbNodeMax < 0 || nbFamily <= 0 || nbFamily > CG_NBMAXFAMILY)
    return 1;
  CloudGraphFree(cloud);
  cloud->_nbNode =
    (int)round(rnd() * (float)(nbNodeMax - nbNodeMin)) + nbNodeMin;
  cloud->_nodes =
    (CloudGraphNode*)malloc(sizeof(CloudGraphNode) * cloud->_nbNode);
  if (cloud->_nodes == NULL) {
    fprintf(stderr, "allocation memory failed\n");
    return 2;
  }
  cloud->_edges =
    (CloudGraphEdge*)malloc(sizeof(CloudGraphEdge) *
      pow(nbNodeMax, 2.0));
  if (cloud->_edges == NULL) {
    fprintf(stderr, "allocation memory failed\n");
    return 2;
  }
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    node->_id = iNode;
    node->_family = (int)round(rnd() * (float)(nbFamily - 1));
    node->_nbLink = 0;
    node->_index = iNode;
  }
  float *connRate = (float*)malloc(sizeof(float) * nbFamily * nbFamily);
  for (int iFamily = 0; iFamily < nbFamily; ++iFamily) {
    for (int jFamily = 0; jFamily < nbFamily; ++jFamily) {
      connRate[iFamily * nbFamily + jFamily] = 0.5 * pow(rnd(), 2.0);
    }
  }
  for (int iNode = 0; iNode < cloud->_nbNode - 1; ++iNode) {
    for (int jNode = iNode + 1; jNode < cloud->_nbNode; ++jNode) {
      CloudGraphNode *nodeA = &(cloud->_nodes[iNode]);
      CloudGraphNode *nodeB = &(cloud->_nodes[jNode]);
      if (rnd() < connRate[nodeA->_family * nbFamily +
        nodeB->_family]) {
        cloud->_edges[cloud->_nbEdge]._nodes[0] = iNode;
        cloud->_edges[cloud->_nbEdge]._nodes[1] = jNode;
        ++(cloud->_nodes[iNode]._nbLink);
        ++(cloud->_nodes[jNode]._nbLink);
        ++(cloud->_nbEdge);
      }
    }
  }
  cloud->_nbFamily = nbFamily;
  free(connRate);
  return 0;
}
```

```
void CloudGraphPrint(CloudGraph *cloud, FILE* stream) {
  if (cloud == NULL || stream == NULL) return;
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    fprintf(stream, "%d %f %f\n", node->_id, node->_pos[0],
      node->_pos[1]);
  }
}

void CloudGraphSetFamilyColor(CloudGraph *cloud, int family,
  unsigned char *rgb) {
  if (cloud == NULL || family >= CG_NBMAXFAMILY || rgb == NULL)
    return;
  cloud->_colors[family * 3] = rgb[0];
  cloud->_colors[family * 3 + 1] = rgb[1];
  cloud->_colors[family * 3 + 2] = rgb[2];
}

void CloudGraphGetFamilyCenter(CloudGraph *cloud, int iFamily,
  float *pos) {
  pos[0] = pos[1] = 0.0;
  if (cloud == NULL || iFamily < 0 || iFamily >= CG_NBMAXFAMILY ||
    pos == NULL)
    return;
  int nb = 0;
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    if (node->_family == iFamily) {
      pos[0] += node->_pos[0];
      pos[1] += node->_pos[1];
      ++nb;
    }
  }
  pos[0] /= (float)nb;
  pos[1] /= (float)nb;
}
```

4 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3 -s
OPTIONS=$(OPTIONS_RELEASE) -lm

all : main

main: main.o CloudGraph.o Makefile /home/bayashi/Coding/TGA/tga.o /home/bayashi/Coding/SpringSys/
	gcc -o main main.o /home/bayashi/Coding/TGA/tga.o  /home/bayashi/Coding/SpringSys/C/Sprin

main.o : main.c /home/bayashi/Coding/TGA/tga.h CloudGraph.h Makefile
	gcc -c main.c $(OPTIONS)

CloudGraph.o : CloudGraph.c CloudGraph.h /home/bayashi/Coding/SpringSys/C/SpringSys.h Makefile
	gcc -c CloudGraph.c $(OPTIONS)

clean :
	rm -rf *.o main
```

5 Usage

The front-end can be executed from the command line. Its arguments are described below:

- `[-tga <filename>]` : name of the TGA file output; if not given, no TGA file is generated.
- `[-print]` : print on the standard output the positions of nodes, one node per line, separated by a space.
- `[-stretch]` : print on the standard output the stretch value of the cloud (0.0 means the constraint on the nodes are completely satisfied, if not satisfied the higher the worst).
- `[-graph <filename>]` : name of the file describing the graph; the format is given below; if not given a random graph is generated.
- `[-circle]` : arrange nodes around a circle; see 1.1 for detail; this is the default arrangement if not precised.
- `[-spring]` : arrange nodes as a spring-mass system; see 1.2 for detail.
- `[-rnd nbNodeMin nbNodeMax nbMaxCategory]` : in case a random graph is generated uses these integer values to set respectively the minimum number of nodes, the maximum number of nodes, the number of categories; if not given the default values are 5 for minimum number of nodes, 50 for maximum number of nodes, and a random value between 1 and 10 for number of categories.
- `[-noFamilyLink]`: when used with the spring option avoid the steps 3 and 4 in subsection 1.2.
- `[-curved]` : curve the lines between nodes; see 1.1 and 1.2 for details.

The format for the definition of the graph is as follow:

- on the first line, two integers: the number of nodes and the number of edges
- on the nbNode following lines, two integer: the id of the node and the id of the family (from 0 to 99)

- on the nbEdge following lines, two integer: the id of the two nodes connected by the edge

Example:

```
3 3
0 0
1 0
2 0
0 1
1 2
2 0
```

Code of the front-end:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "/home/bayashi/Coding/TGA/tga.h"
#include "CloudGraph.h"

void CloudGraphToTGA(CloudGraph *cloud, char *fileName) {
  int sizeNode = 5;
  int size = sizeNode * 10 * (int)round(sqrt(cloud->_nbNode));
  short dim[2];
  dim[0] = dim[1] = size;
  TGAPixel pixA;
  pixA._rgba[0] = pixA._rgba[1] = pixA._rgba[2] = 255;
  pixA._rgba[3] = 255;
  TGAPixel pixB;
  pixB._rgba[0] = pixB._rgba[1] = pixB._rgba[2] = 255;
  pixB._rgba[3] = 255;
  TGA *tga = TGACreate(dim, &pixA);
  short r[2];
  r[0] = r[1] = sizeNode;
  float trans = sizeNode;
  float scale = (float)size * (float)size / (3.0 * trans + (float)size);
  for (int iEdge = 0; iEdge < cloud->_nbEdge; ++iEdge) {
    short from[2];
    short to[2];
    CloudGraphEdge *edge = &(cloud->_edges[iEdge]);
    CloudGraphNode *fromNode = &(cloud->_nodes[edge->_nodes[0]]);
    CloudGraphNode *toNode = &(cloud->_nodes[edge->_nodes[1]]);
    from[0] = (short)round(scale * fromNode->_pos[0] + trans);
    from[1] = (short)round(scale * fromNode->_pos[1] + trans);
    to[0] = (short)round(scale * toNode->_pos[0] + trans);
    to[1] = (short)round(scale * toNode->_pos[1] + trans);
    int family = (fromNode->_family < CG_NBMAXFAMILY ?
      fromNode->_family : CG_NBMAXFAMILY - 1);
    unsigned char *col = &(cloud->_colors[3 * family]);
    pixA._rgba[0] = col[0];
    pixA._rgba[1] = col[1];
    pixA._rgba[2] = col[2];
    family = (toNode->_family < CG_NBMAXFAMILY ?
      toNode->_family : CG_NBMAXFAMILY - 1);
    col = &(cloud->_colors[3 * family]);
    pixB._rgba[0] = col[0];
    pixB._rgba[1] = col[1];
    pixB._rgba[2] = col[2];
```

```
    if (cloud->_mode == CloudGraphModeSpring) {
      if (cloud->_curvedLink == 1) {
        float c[2];
        CloudGraphGetFamilyCenter(cloud, fromNode->_family, c);
        c[0] *= (float)size;
        c[1] *= (float)size;
        if (fromNode->_family != toNode->_family) {
          float cp[2];
          CloudGraphGetFamilyCenter(cloud, toNode->_family, cp);
          cp[0] *= (float)size;
          cp[1] *= (float)size;
          c[0] = 0.5 * (c[0] + cp[0]);
          c[1] = 0.5 * (c[1] + cp[1]);
        }
        short ctrlPt[2];
        ctrlPt[0] = (short)round(c[0]);
        ctrlPt[1] = (short)round(c[1]);
        TGADrawCurve(tga, from, to, ctrlPt, ctrlPt, &pixA, &pixB);
      } else {
        TGADrawLink(tga, from, to, &pixA, &pixB);
      }
    } else if (cloud->_mode == CloudGraphModeCircle) {
      if (cloud->_curvedLink == 0) {
        TGADrawLink(tga, from, to, &pixA, &pixB);
      } else {
        short ctrlPt[2];
        ctrlPt[0] = (short)round(0.5 * (float)size);
        ctrlPt[1] = ctrlPt[0];
        if (fromNode->_family == toNode->_family) {
          float c[2];
          c[0] = 0.5 * (float)(fromNode->_pos[0] + toNode->_pos[0]);
          c[1] = 0.5 * (float)(fromNode->_pos[1] + toNode->_pos[1]);
          c[0] *= (float)size;
          c[1] *= (float)size;
          ctrlPt[0] =
            (short)round(0.75 * c[0] + 0.25 * (float)(ctrlPt[0]));
          ctrlPt[1] =
            (short)round(0.75 * c[1] + 0.25 * (float)(ctrlPt[1]));
        }
        TGADrawCurve(tga, from, to, ctrlPt, ctrlPt, &pixA, &pixB);
      }
    }
  }
  for (int iNode = 0; iNode < cloud->_nbNode; ++iNode) {
    short center[2];
    CloudGraphNode *node = &(cloud->_nodes[iNode]);
    center[0] = (short)round(scale * node->_pos[0] + trans);
    center[1] = (short)round(scale * node->_pos[1] + trans);
    int family = (node->_family < CG_NBMAXFAMILY ?
      node->_family : CG_NBMAXFAMILY - 1);
    unsigned char *col = &(cloud->_colors[3 * family]);
    pixA._rgba[0] = col[0];
    pixA._rgba[1] = col[1];
    pixA._rgba[2] = col[2];
    TGAFillEllipse(tga, center, r, &pixA);
  }
  TGASave(tga, fileName);
  TGAFree(tga);
  free(tga);
}

#define rnd() (float)(rand())/(float)(RAND_MAX)
```

```
int main(int argc, char **argv) {
  time_t seed = time(NULL);
  srandom(seed);
  //srandom(3);
  // Allocate memory for the graph
  CloudGraph *cloud = CloudGraphCreate();
  // Decode arguments
  char flagPrint = 0;
  char flagStretch = 0;
  char *fileNameTGA = NULL;
  char *fileNameGraph = NULL;
  int nbNodeMin = 5;
  int nbNodeMax = 50;
  int nbFamily = 1 + (int)round(rnd() * 9.0);
  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], "-noFamilyLink") == 0) {
      cloud->_familyLink = 0;
    } else if (strcmp(argv[iArg], "-curved") == 0) {
      cloud->_curvedLink = 1;
    } else if (strcmp(argv[iArg], "-circle") == 0) {
      cloud->_mode = CloudGraphModeCircle;
    } else if (strcmp(argv[iArg], "-spring") == 0) {
      cloud->_mode = CloudGraphModeSpring;
    } else if (strcmp(argv[iArg], "-stretch") == 0) {
      flagStretch = 1;
    } else if (strcmp(argv[iArg], "-graph") == 0 && iArg + 1 < argc) {
      fileNameGraph = argv[iArg + 1];
      ++iArg;
    } else if (strcmp(argv[iArg], "-rnd") == 0 && iArg + 3 < argc) {
      nbNodeMin = atof(argv[iArg + 1]);
      nbNodeMax = atof(argv[iArg + 2]);
      nbFamily = atof(argv[iArg + 3]);
      iArg += 3;
    } else if (strcmp(argv[iArg], "-help") == 0) {
      printf("arguments : [-tga <filename>] [-print] [-stretch]");
      printf(" [-graph <filename>] [-circle] [-spring]");
      printf(" [-rnd nbNodeMin nbNodeMax nbMaxCategory]");
      printf(" [-noFamilyLink] [-curved]\n");
    }
  }
  // Read the graph from the input file
  if (fileNameGraph == NULL) {
    // no input file, create a random graph
    int ret = CloudGraphCreateRnd(cloud, nbNodeMin,
      nbNodeMax, nbFamily);
    if (ret != 0) {
      fprintf(stderr,
        "Error while creating the random graph (%d)\n", ret);
      CloudGraphFree(cloud);
      free(cloud);
      return 1;
    }
  } else {
    int ret = CloudGraphLoad(cloud, fileNameGraph);
    if (ret != 0) {
      fprintf(stderr, "Error while loading the graph file (%d)\n", ret);
```

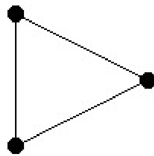
```
      CloudGraphFree(cloud);
      free(cloud);
      return 1;
    }
  }
  // Create the CloudGraph
  CloudGraphArrange(cloud);
  // Save the result in a tga picture
  if (fileNameTGA != NULL) {
    CloudGraphToTGA(cloud, fileNameTGA);
  }
  // Print the stretch of the cloud
  if (flagStretch == 1) {
    float stretch = CloudGraphStretch(cloud);
    printf("Stretch: %f\n", stretch);
  }
  // Print the cloud
  if (flagPrint == 1) {
    CloudGraphPrint(cloud, stdout);
  }
  // Free memory
  CloudGraphFree(cloud);
  free(cloud);
  return 0;
}
```

6 Examples

6.1 testCloud1.txt

Given the following graph definition:

```
3 3
0 0
1 0
2 0
0 1
1 2
2 0
```



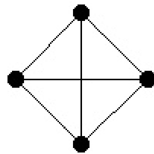
main -tga cloud.tga -graph testCloud1.txt -stretch -print

```
Stretch: 0.236065
0 1.000000 0.499999
1 0.000003 1.000000
2 0.000000 0.000000
```

6.2 testCloud2.txt

Given the following graph definition:

```
4 6
0 0
1 0
2 0
3 0
0 1
0 2
0 3
2 1
2 3
1 3
```



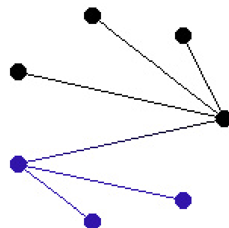
main -tga cloud.tga -graph testCloud2.txt -stretch -print

```
Stretch: 1.171573
0 1.000000 0.500000
1 0.500001 1.000000
2 0.000000 0.500001
3 0.499998 0.000000
```

6.3 testCloud3.txt

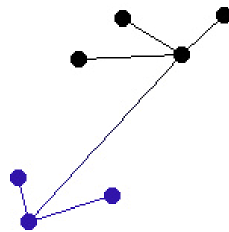
Given the following graph definition:

```
7 6
0 0
1 0
2 0
3 0
4 1
5 1
6 1
0 1
0 2
0 3
0 4
4 5
4 6
```



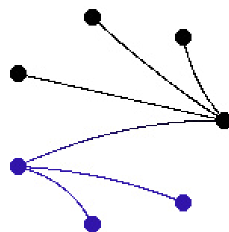
main -tga cloud.tga -graph testCloud3.txt -stretch -print

```
Stretch: 7.371079
0 1.000000 0.500000
1 0.801938 0.900968
2 0.356897 1.000000
3 0.000001 0.722522
4 0.000000 0.277480
5 0.356894 0.000000
6 0.801936 0.099029
```

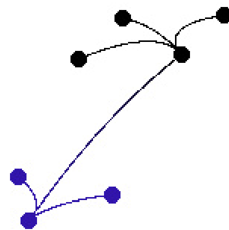


main -tga cloud.tga -graph testCloud3.txt -stretch -print -spring

```
Stretch: 8.614583
0 0.791474 0.803444
1 1.000000 1.000000
2 0.505365 0.980468
3 0.296839 0.783913
4 0.048671 0.000000
5 0.452385 0.126143
6 0.000000 0.215526
```



main -tga cloud.tga -graph testCloud3.txt -curved

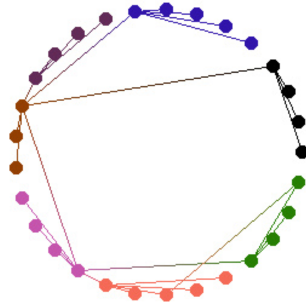


main -tga cloud.tga -graph testCloud3.txt -curved -spring

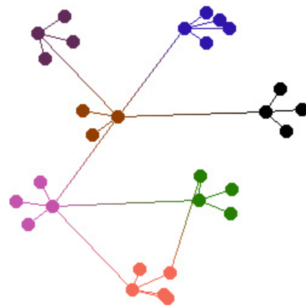
6.4 testCloud4.txt

Given the following graph definition:

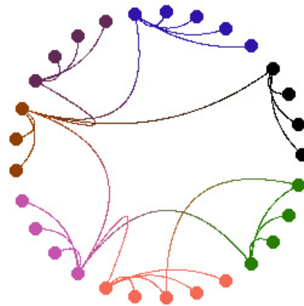
```
29 29
0 0
1 0
2 0
3 0
4 1
5 1
6 1
7 1
8 1
9 2
10 2
11 2
12 2
13 3
14 3
15 3
16 4
17 4
18 4
19 4
20 5
21 5
22 5
23 5
24 5
25 6
26 6
27 6
28 6
0 3
1 3
2 3
3 13
4 8
7 8
5 8
6 8
8 13
15 13
14 13
13 12
13 19
9 12
10 12
11 12
16 19
17 19
18 19
19 25
19 20
21 20
22 20
23 20
24 20
26 25
27 25
28 25
22 28
```


main -tga cloud.tga -graph testCloud4.txt -stretch -print

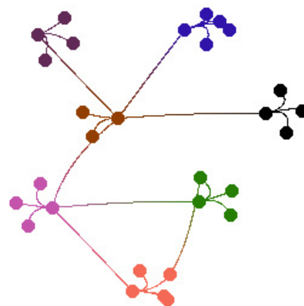
```
Stretch: 71.312012
0 1.000000 0.500000
1 0.988276 0.607643
2 0.953652 0.710253
3 0.897747 0.803031
4 0.823175 0.881641
5 0.733423 0.942405
6 0.632688 0.982482
7 0.525680 1.000000
8 0.417402 0.994138
9 0.312918 0.965171
10 0.217113 0.914453
11 0.134466 0.844356
12 0.068843 0.758156
13 0.023311 0.659886
14 0.000000 0.554140
15 0.000000 0.445862
16 0.023310 0.340116
17 0.068842 0.241846
18 0.134464 0.155646
19 0.217111 0.085549
20 0.312916 0.034830
21 0.417400 0.005862
22 0.525677 0.000000
23 0.632686 0.017517
24 0.733421 0.057594
25 0.823173 0.118358
26 0.897745 0.196967
27 0.953651 0.289745
28 0.988275 0.392355
```


main -tga cloud.tga -graph testCloud4.txt -stretch -print -spring

```
Stretch: 72.935112
0 0.946308 0.583065
1 1.000000 0.662535
2 0.925536 0.733858
3 0.871844 0.654388
4 0.667705 0.894452
5 0.666627 1.000000
6 0.710331 0.976959
7 0.744851 0.947592
8 0.589481 0.946860
9 0.170272 0.986108
10 0.198243 0.900757
11 0.103421 0.841625
12 0.075450 0.926977
13 0.357813 0.634505
14 0.233959 0.658309
15 0.265539 0.573541
16 0.000000 0.339543
17 0.085245 0.405059
18 0.040898 0.256634
19 0.126144 0.322150
20 0.408505 0.029677
21 0.431282 0.102691
22 0.538761 0.087218
23 0.515984 0.014204
24 0.529581 0.000000
25 0.640175 0.342032
26 0.754148 0.382666
27 0.748963 0.295297
28 0.645361 0.429402
```



main -tga cloud.tga -graph testCloud4.txt -curved



main -tga cloud.tga -graph testCloud4.txt -curved -spring

6.5 Random graphs