◊ .1

Dynamic Layout Algorithm to Display General Graphs

Laszlo Szirmay-Kalos
Technical University of Budapest
Budapest, Hungary

◊ INTRODUCTION ◊

Many programs designed for interactive applications, scientific visualization, multimedia etc. should display graphs on the computer screen. The emphasis is usually on the topological relationships, the exact position of the nodes is not important (just for a few of them), but the picture of the graph should be easy to understand by human observers. An easy-to-interpret or "nice looking" graph is expected to:

- be in the center of the screen (or in a given part the world coordinate system if ZOOM/PAN operations are allowed), and expand loosely over the available area.
- contain straight line edges crossing each other as little as possible, and nodes having enough room around them to provide some textual information.
- place strongly coupled node groups in a compact manner and put nodes which have large topological distance far away.
- have a "natural" arrangement (similar to those human observers are used to).

These subjective criteria however are almost impossible to be formulated by mathematical means which could serve as the basis for the definition of a optimality condition and of the elaboration of a computer algorithm trying to find it.

In these poorly defined problems we can rely on heuristics and also on analogies of systems which behave somehow similarly but allow a more rigorous analysis. This gem is based on a combined effort to deal with this problem:

1. It uses an analogous approach applying mechanical system theory (**?**) providing "solution" for the "natural" arrangement problem.
2. It takes advantage of heuristic methods to drive the analogous solution and to add those features which are not reflected by the behaviour of mechanical systems and also to reduce the response time of the algorithm.

ISBN 0-12-XXXXXX-X

$\diamondsuit$

$\diamondsuit$ MECHANICAL SYSTEM ANALOGY $\diamondsuit$

The idea of using a mechanical or physical analogy comes from the fact that the graph is supposed to be arranged "naturally", since this approach makes sure that the layout of the graph will be similar to "everyday" systems. Since physical systems are driven to their energy minimum, in the analogous approach also the energy minimum, or the stable state of the graph model is to be found.

Lets model the graph as a mechanical system, where the nodes correspond to particles while edges, or absence of edges correspond to driving forces among these particles. Let the force be attractive between those pairs of particles which are not very close to each other, while let it be repulsive otherwise. Since objects (or particles in the analogy) are expected to be near if they have an edge between them and far away otherwise, the threshold distance of repulsion is defined according to if the two objects are connected by an edge or not. For connected edges this threshold distance (or constraint) is set to exceed the required minimum radius around an object to provide loose picture and to hold space for some text. For not connected objects it should be much greater depending on the screen size (or on the size of the world coordinate system) and the number of objects. This binary classification can be further refined for weighted graphs, where the constraint can be proportional to the value of the weight.

if A and B are arbitrary objects:

$$constraint(A,B) = constraint_{min} + (weight_{max} - weight(A,B)) * constraint_{scale}$$

Let the repulsing and attracting forces be proportional to the difference of the actual distance of the objects and the constraint.

$$force(A,B) = (constraint(A,B) - distance(A,B))/distance(A,B) * (pos(A) - pos(B))$$

where $pos(A)$ is the actual position vector of object A.

In order to keep the objects inside the working area forces of the "walls" of the area boundaries are also taken into account. Lets define the force of a wall zero if the object is far, proportional to the distance from the wall if it is close (in some margin) but still inside, and proportional with some greater value if it is outside. The direction of this force always points to the inner section of the working area and is perpendicular to the wall.

if A is inside the working area, but not in the margin:

$$force_{wall}(A) = 0$$

if A is in the margin:

$$force_{wall}(A) = (margin_{wall} - distance(A, wall)) * margin\_drive_{wall}$$

$\diamondsuit$

if A is outside:

$$force_{wall}(A) = distance(A, wall) * out\_drive_{wall} + margin_{wall} * margin\_drive_{wall}$$

The particles (or nodes) of the graph, can be fixed to some location or can be moveable ones, which are to be placed by the algorithm. Responding to the resultant force each moveable particle will start to move and keep moving until the system reaches the stability, where the resultant driving force and the speed of all particles are zero. In order to make sure that the system does not oscillate forever, friction being proportional to the speed is introduced.

Considering the fact that in the stable state the energy is minimal, attracting particles (related objects) can be expected to be placed close to each other and repulsing (not related) objects far away, while the total system is somewhere in the middle of the working area.

The stable state can be found by discrete time simulation using the following formulas:

Let the resultant driving force of a given particle be D, the mass m, coefficient of the friction μ, the current speed $v(t)$ and the current position $r(t)$ in time t .
The resultant force:

$$F = D - v(t) * \mu$$

Applying Newton's law the approximate speed and position in time $t + dt$ can be calculated:

$$(v(t + dt) - v(t))/dt = F/m$$

$$(r(t + dt) - r(t))/dt = (v(t) + v(t + dt))/2$$

Using the previous definitions $v(t + dt)$ and $r(t + dt)$ can be expressed:

$$v(t + dt) = (1 - f) * v(t) + I * D$$

$$r(t + dt) = r(t) + (v(t) + v(t + dt)) * dt/2$$

where f is the friction parameter:

$$f = \mu/m * dt$$

and I is the inverse inertia:

$$I = dt/m$$

◊

If the friction parameter and the inverse inertia were small, the system would react to the forces very quickly, which is an advantage in the beginning of the simulation but can result in excessive transients and oscillation around the stability point. To overcome this problem I and f are also time dependent allowing quick stabilization without excessive transients and oscillation:

$$f(t) = friction_{min} + (friction_{max} - friction_{min}) * t/time_{max}$$

$$I(t) = iinertia_{min} + (iinertia_{max} - iinertia_{min}) * t/time_{max}$$

where $time_{max}$ is the maximum response time allowed in an interactive environment.

The simulation keeps going until

1. a stable state has been found (D is zero for all the objects)
2. the transient becomes too big ($D > force_{max}$) (hopefully this never happens)
3. too much time has elapsed without finding a stable state.

◊ INITIAL POSITIONS ◊

The initial arrangement of the simulation is also critical because there are many possible stable states giving better or poorer view, and the necessary simulation time also depends on the careful selection of the initial positions.

In order to find appropriate initial positions a heuristic method is proposed which guarantees that related objects are close while not related objects are far away, and also that the initial positions are not very far from the mechanical stability. The heuristic algorithm places the fix objects first than introduces the moveable objects step-by-step placing them on the mirror point of the center of gravity of already introduced, not-related objects to the center of gravity of the related objects. Having placed a new object, its position is perturbed randomly to avoid solutions corresponding to locally extremal energy when for example all the objects are on a single line. The amplitude of the perturbation decreases as the number of the already placed nodes increases. Finally the node is moved to the inside area if it might have fallen outside, and the algorithm of the mechanical system analogy is applied to find a nearly stable state before introducing the next object.

In the sample program the *Initial Placement* function is responsible for this initial placement of nodes, while the function of mechanical analogy is called *Dynamic Layout*.

◊

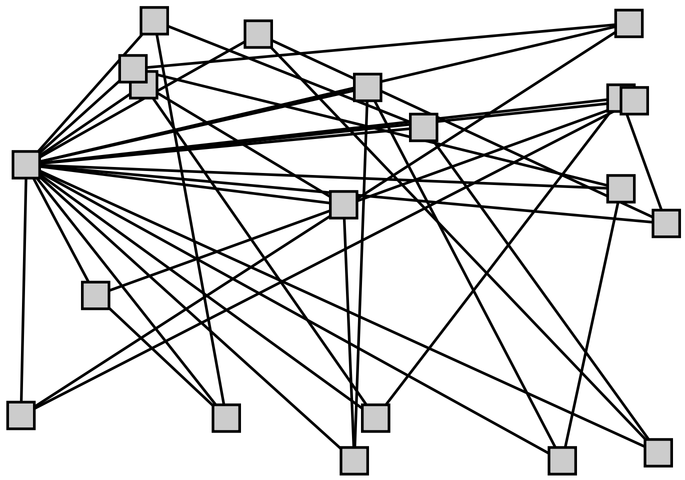

Figure 1. A sample graph arranged randomly

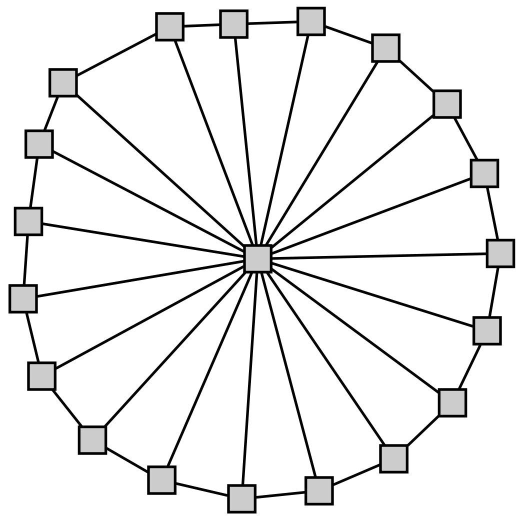

Figure 2. The same graph after running the Algorithm

◊

◊ CONCLUSIONS AND APPLICATIONS ◊

The implementation of the proposed algorithm has been proven to be successful in many applications including modeling of human relationships and information flow in a company, CASE support for object-oriented development etc. Although the algorithm does not take any explicit steps to eliminate the edge crossings it is quite effective in doing so, since it minimizes the length of edges and keeps not related parts together, but far away from the not-related objects which eventually decreases the possibility of edge crossings too.

In interactive systems, where the user supplies and deletes nodes and edges, only the *Dynamic Layout* algorithm is to run to refine the arrangement while sustaining the general outline having been specified by the user. In other cases when the graph should be built from the beginning the *Initial Placement* function has to be called.

Concerning the possible enhancements of the algorithm it is worth mentioning that in many applications the edges are also labelled which makes it necessary to keep away the nodes from the middle points of the edges. This can be achived if the edge centers have also some repulsive force if a node is too close to it. Global control of the layout can be introduced by simulating field forces.

Since the mechanical simulation algorithm goes from the initial position towards the "optimal" arrangement, even if it is stopped before the stabilization due to the response time constraints, a quasi optimal picture can be expected. This allows graphs containing more than 50 vertices to be rendered interactively.

◊

◊ C++ Code ◊

```
/****************************************************************************
/** DYNAMIC LAYOUT ALGORITHM OF GENERAL GRAPHS
/** Author: dr. Szirmay-Kalos Laszlo
/**         szirmay@fsz.bme.hu
/** Date: May 9, 1993, Budapest.
/**     To simplify the notations just those class members are listed which are
/**     actually used in the layout algorithm, while members being necessary
/**     to input the data are missing
/****************************************************************************/
#include <stdlib.h>
#include "defines.h"
#include "vector.hxx"

/*************************************************************************/
class Node {                       // NODE OF THE GRAPH
/*************************************************************************/
    char     name[MAXNAME + 1];    // node name
    TYPE     type;                 // fixed or movable
    vector   pos;                  // actual position
    vector   speed;                // speed
    vector   force;                // driving force to this node
public:
             Node( pchar, TYPE );
    vector&  Position( void )         { return pos;      }
    vector&  Speed( void )            { return speed;    }
    vector&  Force( void )            { return force;    }
    void     AddForce( vector& f )    { force += f;      }
};

/*************************************************************************/
class Relation {                   // EDGE OF THE GRAPH
/*************************************************************************/
    char     name[MAXNAME + 1];    // relation name
    double   intensity;            // relation intensity
    Node   * relation_to;          // related node
public:
             Relation( pchar , Node *, double );
    double   GetRelation( void )     { return intensity;     }
    Node   * GetOtherNode( void )    { return relation_to;   }
};

/*************************************************************************/
class RelationElem : public Relation { // EDGE ITEM IN DYNAMIC DATA STRUCT
/*************************************************************************/
    RelationElem * next_relation;  // next on the list
public:
    RelationElem( pchar name, Node * p, double r );
};
```

◊

```
/*********************************************************************/
class NodeElem : public Node {    // NODE ITEM IN DYNAMIC DATA STRUCT.
/*********************************************************************/
    int            ser_num;        // serial number in list
    NodeElem     * next_node;      // pointer to next node
    RelationElem * relation;       // first relation of this node
public:
                    NodeElem(pchar, TYPE);
    void            SetRelation( RelationElem * p ) { relation = p;          }
    void            SetSerNum( int sernum )         { ser_num = sernum;      }
    NodeElem     *  GetNext( void )                 { return next_node;      }
    RelationElem *  GetRelation( void )             { return relation;       }
    int             GetSerNum( void )               { return ser_num;        }
};

/*********************************************************************/
class Graph { // GRAPH IS AN ORTHOGONAL LIST OF NODES AND EDGES
/*********************************************************************/
    int            nfixnode;         // number of fixed nodes
    int            nmovnode;         // number of moveable nodes
    NodeElem     * currnode;         // current node
    NodeElem     * relatenode;       // actual relation of curr
    RelationElem * currelation;      // relation of nodes list
public:
    BOOL    SearchRelation( void );  // relation currnode and relatenode
    int     InitialPlacement( void );// place nodes step-by-step
    void    RandomArrange( void );   // arrange nodes randomly
    int     DynamicLayout( int );    // mechanical system analogy
    BOOL    FirstNode( void );       // select first node on the list
    BOOL    FirstMoveNode( void );   // select first moveable node
    BOOL    NextNode( int max = ALL_NODES ); // select next to currnode
};

/********************************************************************
*    CONSTANTS
********************************************************************/
const double OVERWINDOW_X   = 1000;    // geometry of the working area
const double OVERWINDOW_Y   = 1000;
const double WALL_MARGIN    = OVERWINDOW_X / 10;
const double TIME_STEP      = 1;       // time step of diff equ
const double MAX_FORCE      = 500;     // force shows instability
const double MIN_FORCE      = 5;       // force cosidered as 0
const double MAX_TIME_SCALE = 100;     // scale of max time of solution
const double MINFRICTION    = 0.6;     // friction boundaries
const double MAXFRICTION    = 0.9;
const double MINIINERTIA    = 0.1;     // inverse inertia boundaries
const double MAXIINERTIA    = 0.4;
const double ZERO_DIST = OVERWINDOW_X / 100;    // distance considered as 0
const double WOUT_DRV       =   80;    // forces of the wall
const double WMARGIN_DRV    =   1;
const double SCALECONSTRAINT = OVERWINDOW_X / 3.5 / MAXRELATION;
const double MINCONSTRAINT  = OVERWINDOW_X / 7; // minimal constraint
```

◊

```
/****************************************************************************/
/* DYNAMIC LAYOUT based on MECHANICAL SYSTEM ANALOGY                        */
/* IN  : The serial number of the maximal moveable node to be considered    */
/* OUT : STOPPED  = All objects stopped                                     */
/*       INSTABLE = Instable, force goes to infinity                        */
/*       TOO_LONG = Too much time elapsed                                   */
/****************************************************************************/
int Graph :: DynamicLayout( int maxsernum )
/*--------------------------------------------------------------------------*/
{
    double constraint, friction, iinertia, dist;
    vector drive;                       // drive forces
    vector direction;                   // direction of drives
    double MAX_TIME = MAX_TIME_SCALE * (nmovnode + nfixnode + 1);

    if ( !FirstMoveNode() ) return STOPPED; // INIT SPEED OF NODES TO 0
    do  currnode->Speed() = vector(0, 0); while ( NextNode(maxsernum) );
/*
*    MAIN CYCLE OF TIME IN THE SOLUTION OF DIFF EQUATION
*/
    for ( double t = 0.0 ; t < MAX_TIME ; t += TIME_STEP ) {

        FirstNode();        //        INITIALIZE FORCE IN NODES TO 0
        do currnode->Force() = vector(0, 0); while( NextNode(maxsernum) );
                            //    CALCULATE FRICTION AND RESPONSE VALUES FROM t
        friction = MINFRICTION + (MAXFRICTION - MINFRICTION) * t / MAX_TIME;
        iinertia = MAXIINERTIA - (MAXIINERTIA - MINIINERTIA) * t / MAX_TIME;
/*
*    CALCULATE DRIVE FORCE BETWEEN EACH PAIR OF NODES
*/
        FirstNode();
        do {
          relatenode = currnode->GetNext();
          while( relatenode != NULL && relatenode->GetSerNum() <= maxsernum ) {
            direction = currnode->Position() - relatenode->Position();
            dist  = direction.Size();
            if ( dist < ZERO_DIST ) dist = ZERO_DIST;
            switch( SearchRelation() ) {  // CALCULATE FORCE FROM THEIR RELATION
            case NOT_FOUND:
                constraint = MINCONSTRAINT + MAXRELATION * SCALECONSTRAINT;
                break;
            case FOUND:
                constraint = MINCONSTRAINT +
                     (MAXRELATION - currelation->GetRelation()) * SCALECONSTRAINT;
                break;
            }
            drive = (constraint - dist) / dist * direction; // SET FORCE
            drive /= (double)(maxsernum + nfixnode);
            currnode->AddForce(drive);
            relatenode->AddForce(-drive);
            relatenode = relatenode->GetNext();
          }
        } while ( NextNode( maxsernum ) );
```

◊

```
/*
*    ADD FORCES OF THE WALLS AND DETERMINE MAXIMAL FORCE
*/
        double max_force = 0.0;
        FirstMoveNode();
        do {
           dist = currnode->Position().X();
           if (dist < 0) {                     // OUT LEFT
             drive = vector(-dist * WOUT_DRV + WALL_MARGIN * WMARGIN_DRV, 0);
             currnode->AddForce(drive);
           } else if (dist < WALL_MARGIN) {    // IN LEFT MARGIN
             drive = vector((WALL_MARGIN - dist) * WMARGIN_DRV, 0);
             currnode->AddForce(drive);
           }

           dist = currnode->Position().X() - OVERWINDOW_X;
           if (dist > 0) {                     // OUT RIGHT
             drive = vector(-dist * WOUT_DRV + WALL_MARGIN * WMARGIN_DRV, 0);
             currnode->AddForce(drive);
           } else if (-dist < WALL_MARGIN) {   // IN RIGHT MARGIN
             drive = vector((-WALL_MARGIN - dist) * WMARGIN_DRV, 0);
             currnode->AddForce(drive);
           }

           dist = currnode->Position().Y();
           if (dist < 0) {                    // OUT BOTTOM
             drive = vector(0, -dist * WOUT_DRV + WALL_MARGIN * WMARGIN_DRV);
             currnode->AddForce(drive);
           } else if (dist < WALL_MARGIN) {   // IN BOTTOM MARGIN
             drive = vector(0, (WALL_MARGIN - dist) * WMARGIN_DRV);
             currnode->AddForce(drive);
           }

           dist = currnode->Position().Y() - OVERWINDOW_Y;
           if (dist > 0) {                    // OUT TOP -> FORCE OF TOP WALL
             drive = vector(0, -dist * WOUT_DRV + WALL_MARGIN * WMARGIN_DRV);
             currnode->AddForce(drive);
           } else if (-dist < WALL_MARGIN) {  // IN TOP MARGIN
             drive = vector(0, (-WALL_MARGIN - dist) * WMARGIN_DRV);
             currnode->AddForce(drive);
           }
/*
*    NEWTON'S LAW -> STEP TO t + dt
*/
            vector old_speed = currnode->Speed( );
            currnode->Speed() =
                  (1 - friction) * old_speed + iinertia * currnode->Force();
            currnode->Position() +=
                  0.5 * TIME_STEP * (old_speed + currnode->Speed() );

            double abs_force = currnode->Force().Size( ); // CALC. MAX FORCE
            if ( abs_force > max_force) max_force = abs_force;
        } while ( NextNode( maxsernum ) );
```

◊

```
/*
*    STOP CALCULATION IF
*/
        if ( max_force < MIN_FORCE ) return STOPPED;  // All objects stopped
        if ( max_force > MAX_FORCE ) return INSTABLE; // Too big transient
    }
    return TOO_LONG; // Too much time elapsed
}


/*******************************************************************/
/* INITIAL PLACEMENT ALGORITHM                                     */
/* OUT : STOPPED  = All objects stopped                            */
/*       INSTABLE = Instable, force goes to infinity               */
/*       TOO_LONG = Too much time elapsed                          */
/*******************************************************************/
int Graph :: InitialPlacement( )
/*-----------------------------------------------------------------*/
{
    vector      candidate; // candidate position
    vector      relate_cent;            // center of related nodes
    vector      notrel_cent;            // center of not related nodes
    vector center(OVERWINDOW_X / 2, OVERWINDOW_Y / 2); // centre point
    int         nrel;                   // number of related nodes
    int         nnotrel;                // number of not related nodes
    double      perturb_x = OVERWINDOW_X / (double)RAND_MAX ;
    double      perturb_y = OVERWINDOW_Y / (double)RAND_MAX ;

    if ( !FirstMoveNode() ) return STOPPED;     //  SKIP FIXED NODES
/*
*    MAIN CYCLE OF INTRODUCING MOVEABLE NODES STEP-BY-STEP
*    CALCULATE THE CENTER OF GRAVITY OF ALREADY INTRODUCED NODES
*    relate_cent IS FOR RELATED NODES
*    notrel_cent IS FOR NON_RELATED NODES
*/
    for( int inode = 1; ; inode++ ) {
        relate_cent = notrel_cent = vector(0, 0);
        nrel = nnotrel = 0;                // displayed nodes
        relatenode = currnode;

        for( FirstNode(); currnode != relatenode; NextNode() ) {
            switch ( SearchRelation() ) {
            case NOT_FOUND:  notrel_cent += currnode->Position();
                             nnotrel++;
                             break;
            case FOUND:      relate_cent += currnode->Position();
                             nrel++;
                             break;
            }
        }

        if ( nrel != 0 )      relate_cent /= (double)nrel;
        if ( nnotrel != 0 )  notrel_cent /= (double)nnotrel;
```

◊

```
/*
*    IF THIS IS THE FIRST POINT->PUT TO THE MIDDLE
*/
        if (nrel == 0 && nnotrel == 0) candidate = center;
        else
/*
*    IF NO NOT_RELATED NODE -> PUT TO THE CENTRE OF GRAVITY OF RELATED NODES
*/
        if ( nnotrel == 0 ) candidate = relate_cent;
        else
/*
*    IF NO RELATED NODE->PUT TO THE MIRROR OF THE nrel_cent ON THE CENTRE
*/
        if ( nrel == 0 )  candidate = 2.0 * center - notrel_cent;
        else
/*
*   BOTH TYPE OF NODES EXIST ->
*   CALCULATE THE CANDIDATE POINT AS THE MIRROR OF notrel_cent TO relate_cent
*/
            candidate = 2.0 * relate_cent - notrel_cent;
/*
*    PERTURBATE RANDOMLY
*/
        candidate += vector( perturb_x / (double)(nfixnode + inode + 5) *
                             (double)( rand() - RAND_MAX / 2),
                             perturb_y / (double)(nfixnode + inode + 5) *
                             (double)( rand() - RAND_MAX / 2 ));
/*
*    DECIDE IF IT IS OUTSIDE->FIND THE NEAREST INSIDE POINT
*/
        if ( candidate.X() < WALL_MARGIN)
            candidate = vector( 2 * WALL_MARGIN, candidate.Y() );
        if ( candidate.X() > OVERWINDOW_X - WALL_MARGIN)
            candidate = vector(OVERWINDOW_X - 2 * WALL_MARGIN, candidate.Y());
        if ( candidate.Y() < WALL_MARGIN)
            candidate = vector( candidate.X(), 2 * WALL_MARGIN );
        if ( candidate.Y() > OVERWINDOW_Y - WALL_MARGIN)
            candidate = vector(candidate.X(), OVERWINDOW_Y - 2 * WALL_MARGIN );

        relatenode->Position( ) = cand
idate; // SET POSITION OF THE NEW NODE
/*
*    ARRANGE ALREADY DEFINED NODES BY DYNAMIC LAYOUT
*/
        NodeElem * oldcurrnode = currnode;
        int ret = DynamicLayout( inode );
        currnode = oldcurrnode;
        if ( ret != STOPPED || !NextNode() ) return ret;
    }
}
```

Bibliography

(X. Pintado 1991) D. Tsichritzis X. Pintado. Fuzzy relationships and affinity links. In Dennis Tsichritzis, editor, *Object Composition*, pages 273–285. Universite de Geneve, Geneva, 1991.

æææ

ISBN 0-12-XXXXXX-X