University of Arizona, Department of Computer Science

CSc 372 — Assignment 8 — Due midnight, Wed Dec 7 — 10%

Christian Collberg
November 30, 2005

1 Introduction

The purpose of this assignment is improve your Icon programming skills. In particular, you will be using Icon tables, lists, and sets, and do parsing with Icon's string scanning procedures.

The program you will build will display arbitrary undirected graphs in a "nice" way using an iterative algorithm. Graphs are common in many applications which makes such graph drawing programs very useful. See, for example, `http://www.graphviz.org` which is installed on `lectura` as the `dot` program.

We will be grading on correctness and style. Every procedure should be adequately formatted and documented.

This assignment can be worked in teams of two.

2 A Graph Module [30 points]

The first step is to create a module `graph.icn` which will allow us to create new graphs. It has routines for creating an empty graph, adding nodes to this graph, traversing the nodes and edges of the graph, and printing it out. The routines that you need to implement are:

```
record GRAPH(nodes, edges)

procedure graph_create(n)
procedure graph_nodecount(g)
procedure graph_nodes(g)
procedure graph_edges(g, v)
procedure graph_addedge(g, u, v)
procedure graph_show(g)
```

Here is how you would use these routines to create a new graph:

```
g1 := graph_create(5)
graph_addedge(g1, 1, 2)
graph_addedge(g1, 2, 3)
graph_addedge(g1, 3, 4)
graph_addedge(g1, 4, 5)
```

Here's how you'd get the number of nodes, traverse the nodes and edges, and print out the graph:

```
write("Number of nodes: ", graph_nodecount(g1));
write("Outgoing edges from node 2:")
every i := graph_edges(g1, 1) do writes(" ", i)
write("\nNodes:")
every i := graph_nodes(g1) do writes(" ", i)
write("")
graph_show(g1)

g6 := graph_read("g2.gdl")
graph_show(g6)
```

This testing code can be found in the file `testgraph.icn`, and the output should look like this:

```
> testgraph
Number of nodes: 5
Outgoing edges from node 2:
 2
Nodes:
 1 2 3 4 5
1: 2
2: 1 3
3: 2 4
4: 3 5
5: 4

Graph read from g2.gdl:
1: 2 3
2: 1 3
3: 1 2
```

`graph_read(file)` reads a graph from a file in the following format:

```
3
1--2
1--3
2--3
```

The first number is the number of nodes. On subsequent lines the edges are listed, one per line. `graph_read(file)` should be implemented using Icon's *string scanning* functions.

The graph should be represented as a record

```
record GRAPH(nodes, edges)
```

where `nodes` is simply a list of the nodes $[1, 2, \ldots, n]$. `edges` is a table indexed by node number. For every node i, `edges[`i`]` is a set of the outgoing edges from i. For example, if the graph contains the edges $1 \rightarrow 2, 1 \rightarrow 3$, then `edges[`$i$`]` should be the set $\{2, 3\}$.

A template file `graph.icn` will be provided for you. The comments give more details with respect to implementation and error handling.

3 A Graph Generation Module [30 points]

To have something cool to display, we next write the module `graphgen.icn` which generates common graph types. We write routines to generate `list`, `cycle`, `star`, `complete`, and `binary tree` graphs. The list graph routine has been give to you, the rest you have to write yourselves:

```
# A simple link graph where every node n is connected
# to node n+1.
procedure graphgen_List(n)
   local g
   g := graph_create(n)
   every i := 1 to n-1 do
      graph_addedge(g, i, i+1)
   return g
end

# A cycle graph where every node n is connected
# to node n+1, and the last node is connected to
# the first.
procedure graphgen_Cycle(n)

# A star graph where node number 1 is connected to
# every other node.
procedure graphgen_Star(n)

# A complete graph where every node is connected to
# every other node.
procedure graphgen_K(n)

# A binary tree graph where every node n is connected
# to nodes 2n and 2n+1, provided these exist.
procedure graphgen_BinTree(n)
```

There's a test program `testgraphgen.icn` given to you that prints out each graph type:

```
   write("\nList graph:")
   g1 := graphgen_List(5)
   graph_show(g1)

   write("\nCycle graph:")
   g2 := graphgen_Cycle(5)
   graph_show(g2)

   write("\nStar graph:")
```

```
   g3 := graphgen_Star(5)
   graph_show(g3)

   write("\nK(5) graph:")
   g4 := graphgen_K(5)
   graph_show(g4)

   write("\nBinTree graph:")
   g5 := graphgen_BinTree(5)
   graph_show(g5)
```

The output should be like this:

```
> make testgraphgen
> testgraphgen

List graph:
1: 2
2: 1 3
3: 2 4
4: 3 5
5: 4

Cycle graph:
1: 2 5
2: 1 3
3: 2 4
4: 3 5
5: 1 4

Star graph:
1: 2 3 4 5
2: 1
3: 1
4: 1
5: 1

K(5) graph:
1: 1 2 3 4 5
2: 1 2 3 4 5
3: 1 2 3 4 5
4: 1 2 3 4 5
5: 1 2 3 4 5

BinTree graph:
1: 2 3
2: 1 4 5
3: 1
4: 2
5: 2
```

A template file `graphgen.icn` will be provided for you.

4 A Graph Drawing Module [40 points]

The final module is `gdraw.icn` which does the actual graph layout. It will be using the `point.icn` module from the last assignment.

The basic idea of the algorithm can be seen from the main routine, which will be given to you:

```
procedure gdraw_layout(g, W, L, iters, tweak, delay)
   local area, pos, disp, temperature, i

   # Make tweak global so we don't have to carry it
   # around everywhere.
   TWEAK := tweak

   # Will become smaller with each iteration so
   # that changes to the node positions will be
   # less and less.
   temperature := W/10

   # Start with all nodes at random positions.
   pos := point_randomList(graph_nodecount(g),W,L)

   every i := 1 to iters do {
      rdisp := gdraw_repulsiveForce(g, pos, W, L)
      adisp := gdraw_attractiveForce(g, pos, W, L)
      pos := gdraw_positionNodes(g, pos, rdisp, adisp, W, L, temperature)
      temperature -:= W/10/iters
      if delay>0 then
          graph_draw(g, pos, W, L, delay)
   }
   return pos
end
```

The algorithm is iterative. It starts out by placing each node in a random position. In each iteration it computes two forces acting on each node. These are given in the variables `rdisp` and `adisp` (both are lists of `POINTs`, one per node). `rdisp[i]` is the *repulsive* force acting on node `i`. `adisp[i]` is the *attractive* force acting on node `i`. Nodes that are attached with an edge attract each other. Nodes which are not attached to each other try to push each other away. `temperature` gets smaller with each iteration, making the movements of the nodes smaller and smaller, as they settle in on their final positions.

Your job is to implement the routines

```
procedure gdraw_repulsiveForce(g, pos, W, L)
procedure gdraw_attractiveForce(g, pos, W, L)
procedure gdraw_positionNodes(g, pos, rdisp, adisp, W, L, temp)
```

`gdraw_repulsiveForce(g, pos, W, L)` takes the graph `g` and the current position `pos` as argument. `pos` is a a list of `POINT`s, one per node, giving the current location of each node.`disp` is a a list of `POINT`s, one per node, giving the current location of each node.

```
set disp_v to (0,0) for all nodes v
for every node v do
   for every node u ≠ v do
      Δ ← pos_v − pos_u
      Δ_normal ← normal(Δ)
      if Δ_normal is < 0.01 set it to 0.01
      disp_v ← disp_v + Δ/Δ_normal * fr(g, W, L, Δ_normal)
return disp
```

The `normal` routine is already implemented in the `point` module. `fr` is a "magic" function that is given to you in `gdraw.icn`.

`gdraw_attractiveForce(g, pos, W, L)` takes the same arguments and returns the same type of result as `gdraw_repulsiveForce`. Here's the pseudo-code:

```
set disp_v to (0,0) for all nodes v
for every node v do
   for every edge (v, u) do
      Δ ← pos_v − pos_u
      Δ_normal ← normal(Δ)
      if Δ_normal is < 0.01 set it to 0.01
      disp_v ← disp_v − Δ/Δ_normal * fa(g, W, L, Δ_normal)
      disp_u ← disp_u + Δ/Δ_normal * fa(g, W, L, Δ_normal)
return disp
```

`fa` is another "magic" function that is given to you in `gdraw.icn`.

Finally, we're implementing the function `gdraw_positionNodes(g, pos, rdisp, adisp, W, L, temp)`. It takes the old position of the nodes `pos` and computes new ones, based on what `gdraw_repulsiveForce` and `gdraw_attractiveForce` returned (`rdisp` and `adisp`). `temp` is the current "temperature" (how much we want the nodes to move), a real number. Here's the pseudo-code:

```
disp ← rdisp + adisp
for every node v do
    pos_v ← pos_v + disp_v/normal(disp_v) * temp
    x ← min(W/2, max(−W/2, pos_v.x))
    y ← min(L/2, max(−L/2, pos_v.y))
    pos_v ← (x, y)
return pos
```

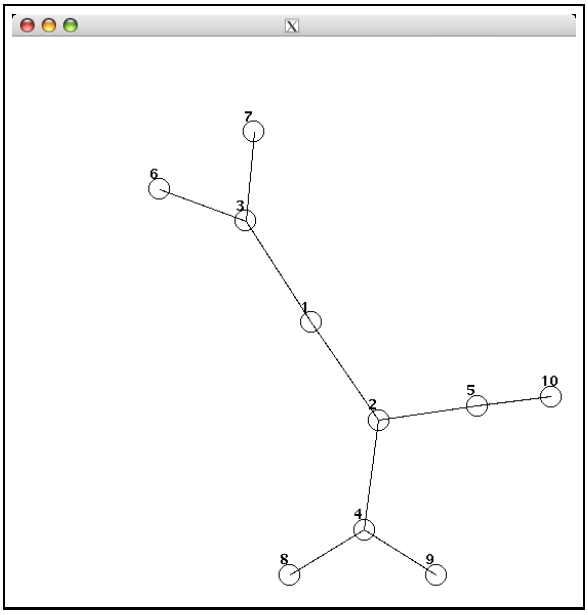
The first line simply adds the results of `gdraw_repulsiveForce` and `gdraw_attractiveForce` together. (We could tweak this, if we wanted to, by weighting the addition (multiplying `adisp` with some constant to make the drawing prettier), but we won't.) The two lines with `min` and `max` in them simply make sure that the new x and y coordinates are inside the drawing window.

5 The Main Program

The main program has been written for you. It simply parses the command line, generates a graph (or reads one from a file), and invokes the graph layout routine. Here is an example:

```
> main -g tree -n 10 -i 50 -d 0
```

`-g tree -n 10` specifies to generate a binary tree with `10` nodes. `-i 50` tells the program to run 50 iterations and `-d 0` to not display intermediate graphs. The result should look like this:



`main -h` will print out all options.

6 Submission and Assessment

The deadline for this assignment is midnight, Wed Dec 7. It is worth 10% of your final grade.

You should submit the assignment electronically using the `Unix` command

```
turnin cs372.8 point.icn graph.icn main.icn graphgen.icn gdraw.icn makefile README
```

.

The `README` file should give the members of your team.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.