

# An Incomplete Reference to the Dominating Set Simulation Suit (*ds*)

Fei Dai (dai2050@yahoo.com)

March 17, 2002

## 1 Introduction

The project *ds* is the first part of my wireless routing simulation suit<sup>1</sup>. *ds* is command line tool which support the simulations of various properties of dominating sets constructed via Dr. Jie Wu's marking process, MCDS, and several cluster-based algorithms. Furthermore, *ds* is written with C++ and highly modulized, therefore, the basic components, like node set, graph, area, etc, are coded as self-containing C++ classes and can be used for other purposes. Originally, *ds* is compiled with GNU C++ (*gcc*) under UNIX platform, but recently, it is transplanted to the WIN32 platforms (Windows 98, NT, 2000, etc), and can be compiled by both *gcc* (under CygWin) and Microsoft Visual C++ (*VC*).

This document is my first attempt to help those others who want to take advantage from the codes already there in *ds*. Because most of them only want to use the basic facilities like random graph generation and graph connectivity test. I'll focus on the usage of the corresponding classes. If you have any further requirement, please let me know<sup>2</sup>.

The rest of this document is organized as follows: Section 2 Fast Start gives a simple example of generating a random graph with classes provided by *ds*, as well as instructions on how to install the *ds* package. Section 3 Class Reference describes several useful *ds* classes and their important methods.

## 2 Fast Start

### 2.1 Install the *ds* package

**Download source files** The latest version of *ds* is 1.1.1. You can download the gzipped tarball (`ds-src-1.1.1.tar.gz`) from either my personal web site (<http://www.cse.fau.edu/~fdai/adhoc>) or the WRSS web site in SourceForge (<http://sourceforge.net/projects/wrss/>).

---

<sup>1</sup><http://sourceforge.net/projects/wrss>

<sup>2</sup><mailto:dai2050@yahoo.com>

**For UNIX users** After downloaded the source package, you can extract it with the commands `gunzip` and `tar`:

```
gunzip -c ds-src-1.1.1.tar.gz | tar -xvf -
```

Then you can use command `make` to build the executable binary file and test it:

```
cd ds-1.1.1/src
make dep
make
ds
```

You will see the simulation result of an ad hoc network with 20 nodes.

**For Visual C++ users** You still need a UNIX environment to unpack the source files. Furthermore, you need to convert the suffix of the “.hh” and “.cc” files into “.h” and “.cpp” so that Visual C++ can correct process them. To do this, in the UNIX environment, type:

```
cd ds-1.1.1/src
make vc
```

You will see a new fold “msvc”, which contains all the “.h” and “.cpp” files. Add all the “.cpp” files into a VC project, build the project and run the program, you should see the same result as in UNIX.

## 2.2 A simple example

The following sample program `sample.cc` is also in the source file package of *ds*. You can compile this program with “`make sample`” on UNIX platform, or build a project with VC including modules *set*, *graph*, *area*, *queue*, and *heap*.

```
#include <time.h>
#include "area.hh"

int main (int argc, char ** argv)
{
    printf ("\nA simple example of ds package Ver 1.0, by Fei Dai, 2002\n\n");

    srand (time(NULL));
```

```

NodeSet::init ();

Area    area (10);
Graph   graph(10);

try {
    printf ("\nPutting 10 wireless devices in a 100X100 area...\n");
    area.shuffle (10);
    printf ("Distribution of devices:\n");
    area.print ();

    printf ("\nGenerating a unit disk graph with transmitter range 50...\n");
    area.generateGraph (graph, 50);
    printf ("Result undirected graph:\n");
    graph.print ();

    printf ("\nTesting graph connectivity...\n");
    if (graph.isConnected (10))
        printf ("The result graph is connected.\n");
    else
        printf ("The result graph is not connected.\n");
}
catch (const char errMsg[]) {
    printf ("Error: %s\n", errMsg);
    return 1;
}

return 0;
}

```

In the above program, “`srand (time(NULL))`” is optional, which resets the random number generator. “`NodeSet::init ()`” is mandatory, which initialize the NodeSet class. Each class may cast exceptions of type string (`const char errMsg[]`), in which case, the “`catch`” facility will print the error message and stop the program.

The generation of a random unit disk graph takes two steps: (1) method “`Area::shuffle(n)`” put  $n$  nodes in a  $100 \times 100$  square, and (2) method “`Area::generateGraph(g, r)`” generate a graph  $g$ , where any two nodes with distance less than  $r$  have a wireless link between them. The “`print`” methods of classes “`Area`” and “`Graph`” print verbose information from debugging purpose.

## 3 Class Reference

### 3.1 Node set

Class *NodeSet* (defined in `set.hh`) is similar to the *bitset* of C++ STL. In *ds*, *NodeSet* stores edges in a graph and nodes in a dominating set.

#### 3.1.1 Declaration

Suppose a node set *ns* is to contain at nodes from 1 to 100, it can be declared as

```
NodeSet ns;  
ns.setSize(100);
```

or

```
NodeSet ns(100);
```

or

```
NodeSet s(100);  
NodeSet ns(s);
```

#### 3.1.2 Access members

Given a node set *ns*, `ns.isMember(x)` or `ns[x]` returns 1 if  $x \in ns$ , 0 otherwise. `ns.memberCount()` returns  $|ns|$ . `ns.isEmpty()` returns 1 if  $ns = \emptyset$ , 0 otherwise. `ns.add(x)` or `ns+=x` sets  $ns = ns \cup \{x\}$ . `ns.remove(x)` or `ns-=x` sets  $ns = ns - \{x\}$ .

To enumerate every members in *ns*, a simple way is

```
for (int i=1; i<n; i++)  
    if ns[i] then {/* do something */}
```

Another more efficient way is

```
for (int i=ns.firstMember(); i; i=ns.nextMember(i))  
    {/* do something */}
```

### 3.1.3 Set computation

Suppose  $ns$  and  $s$  are both nodes sets, `ns.union(s)`, `ns+=s` or `ns|=s` sets  $ns = ns \cup s$ . `ns.minus(s)` or `ns-=s` sets  $ns = ns - s$ . `ns.intersect(s)` or `ns&=s` sets  $ns = ns \cap s$ . `ns.different(s)` or `ns^=s` sets  $ns = (ns - s) \cup (s - ns)$ .

### 3.1.4 Dump information

Suppose  $ns$  is a node set, `ns.print()` prints the list of members of  $ns$ .

## 3.2 Graph

Class *Graph* (defined in `graph.hh`) contains  $n$  nodes, each nodes  $v$  has two node sets: set *absorbants* =  $\{u | (v, u) \in E\}$  contains its downstream neighbors and set *dominants* =  $\{u | (u, v) \in E\}$  contains its upstream neighbors. If  $u$  is an dominant of  $v$ ,  $v$  must be an absorbant of  $u$ , and there is an edge  $(u, v)$  from  $u$  to  $v$ . A link  $(u, v)$  is a bidirectional edge  $((u, v) \in E$  and  $(v, u) \in E)$ .

### 3.2.1 Declaration

Suppose a graph  $g$  is to have at most 100 nodes, it can be declared as

```
Graph g(100);
```

or

```
Graph graph(100);  
Graph g(graph);
```

### 3.2.2 Access nodes and edges

Given a graph  $g$ , `g.isEdge(u,v)` returns 1 if  $(u, v) \in E$ , 0 otherwise. `g.isLink(u,v)` returns 1 if  $(u, v) \in E$  and  $(v, u) \in E$ , 0 otherwise. `g.absorbants(v)` returns  $\{u | (v, u) \in E\}$ . `g.dominants(v)` returns  $\{u | (u, v) \in E\}$ . `g.nodeCount()` returns  $|V|$ . `g.neighborCount(v)` returns  $|\{u | (u, v) \in E \vee (v, u) \in E\}|$ . `g.linkCount()` returns  $|\{(u, v) | (u, v) \in E \vee (v, u) \in E\}|$

`g.addEdge(u,v)` sets  $E = E \cup \{(u, v)\}$ . `g.removeEdge(u,v)` sets  $E = E - \{(u, v)\}$ . `g.addLink(u,v)` sets  $E = E \cup \{(u, v), (v, u)\}$ . `g.neighborSet(ns, v, k)` sets  $ns$  to the  $k$ -hop neighbor set of node  $v$ .

### 3.2.3 Graph connectivity

Given a graph  $g$ , `g.isConnected(u,v)` returns 1 if  $v$  is reachable from  $u$  in  $g$ , 0 otherwise.

`g.isConnected(u,ds)` returns 1 if every  $v \in ds$  is reachable from  $u$  in  $g$ , 0 otherwise.

`g.isConnected(ss,ds)` returns 1 if every  $v \in ds$  is reachable from every  $u \in ss$  in  $g$ , 0 otherwise.

`g.isConnected(n)` returns 1 if every node (from 1 to  $n$ ) is from every node in  $g$ , 0 otherwise.

### 3.2.4 Dump information

Given a graph  $g$ , `g.print()` prints the list of nodes of  $g$ , and for each node, its adjacent neighbors.

## 3.3 Area

Class *Area* (defined in `area.hh`) defines a rectangle dimension  $(X \times Y)$  containing  $n$  wireless devices, each device has position within this rectangle.

### 3.3.1 Declaration

Suppose a  $100 \times 100$  area  $a$  contains at most 100 devices, it can be declared as

```
Area a;
```

or

```
Area a(100);
```

or

```
Area a(100,100,100);
```

The first argument is the maximum number of wireless devices, the second and the third are the  $X$  and  $Y$  dimension of the area.

### 3.3.2 Generate random graph

Suppose  $a$  is an  $100 \times 100$  area with maximum devices number 100, `a.shuffle(100)` randomly places 100 devices in the  $100 \times 100$  rectangle.

`a.generateGraph(g,25)` generate a unit disk graph and store it in  $g$ . In this graph, there is a link between two nodes  $u$  and  $v$  if and only if the geographical distance between the two corresponding wireless devices is within the wireless transmitter range 25.

### 3.3.3 Dump information

Given an area  $a$ , `a.print()` prints the list of wireless devices and their  $(x, y)$  coordinations.

`a.exportNodes("area", "1.nod")` save the positions of wireless devices in  $a$  to file `area/1.nod`. `a.exportArcs("area", "1.arc", "1,ar1", g)` save the start and stop positions of each wireless links in  $g$  into two files. Bidirectional links are stored in `area/1.arc` and unidirectional links are stored in `area/1.ar1`. With these files, you can draw a network graph with *GNUPlot* command

```
plot "1.nod" t "wireless devices", "1.arc" t "wireless links" w l
```