BISON
IST-2001-38923

Biology-Inspired techniques for
Self Organization in dynamic Networks

Readme for Dresden Immune Search Package (disearch)

Classification: Public
Contact Author: Niloy Ganguly
email: niloy@zhr.tu-dresden.de

Document Version: version 1.0 (February 24, 2005)

Project funded by the European Commission under the Information Society Technologies Programme of the 5th Framework (1998-2002)

Contents

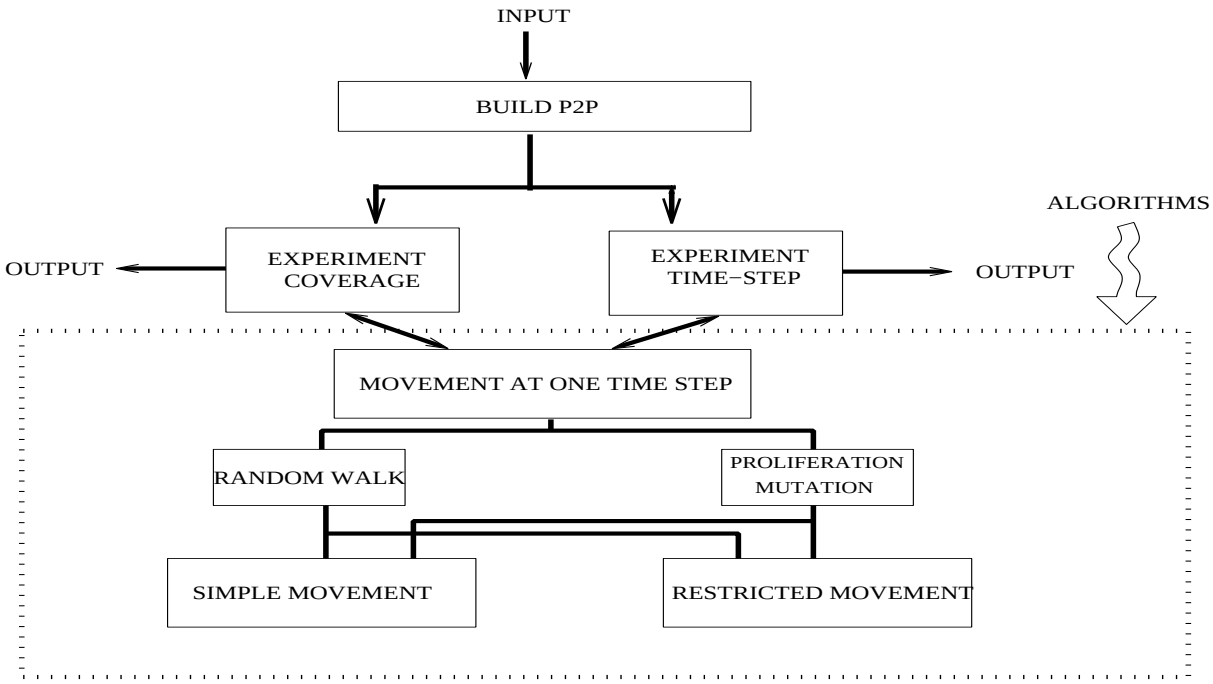

Figure 1: General flow diagram of the program and the position of the different modules in the flow.

1 Implementation details - disearch (Dresden immune search) package

This document details the implementation of the scheme. The scheme is implemented in the Linux environment, although technically it should run in any other environment (with minor modifications). The language used to implement is C.

The program consists of two major modules.

- Model building. Here we build up the desired $p2p$ network. Model building involves three major tasks, namely A. topology determination, B. data and query distribution, and C. determining the sequence of updating.
- Experiments. The experiments *coverage* and *time-step* implement different types of search algorithms[1].

Fig. 1 shows the general flow diagram of the program and the position of the different modules in the flow.

- In the beginning the $p2p$ network is instantiated.
- The desired experimental setup (either coverage or time-step) is chosen.

[1]Kindly note, in this section 'algorithm' essentially refers to the message-forwarding algorithms like random-walk, proliferation-mutation etc.

- The algorithms (random walk or proliferation-mutation) are executed.
- Note, the two experimental setups only differ in terms of the exit condition and output requirements. However, they execute the same algorithms. So, the program design is modularized in terms of one time step.
- The algorithms form two broad categories - the restricted version and the simple version.

The principal modules are now discussed one by one.

1.1 Building up the p2p network

Building the *p2p* network involves construction of the network topology, formation of the data and query distributions, and determination of the sequence in which nodes operate. Therefore, the module implements the following functions.

1. Forming the topology from a given connection list among peers.
2. Constructing content (data) for peers.
3. Building query list.
4. Setting update sequence.

1. Forming the topology

The most important task pertaining to topology formation is the assignment of neighbors of each peer. While assigning the neighbors, it should be assured that the resulting network is connected. A network is connected when there is at least a path connecting two randomly chosen nodes. The neighborhood assignment is obtained from a provided list of connections among peers. This list is generated by external *topology generators* such as BRITE [2] or Inet [1] and also internal grid *topology generators*. The function is defined as below.

```
GenerateNeighborhood(list,p2p)
Input : List of connections among peers. It is a file
                  of the format described in  Fig. 2
Output : Peers containing neighborhood information.
```

The file structure of the topology generator is discussed through *Fig. 2*. The lines starting with '#' are lines which do not appear in the files but are added for the purpose of explanation.

2. Constructing contents of the peers

The second function takes care of building the data base of the peers. We have considered two types of data representation, *simple* and more *realistic*. We have also assured that the distribution of tokens (keywords) in both of them follows Zipf's law.

The simple data representation is accomplished by the following functions.

```
1  102
# <Line No.> <Total number of connections in the network>
2  4 19
# 4 19 means that there is a connection between peer
# number 4 and peer number 19
3  3 200
    .
    .
    .
102 120 19
103 12 197
# Note GenerateNeighborhood reads in only unique relations, that is,
# if there is information about connection between 4 19,
# then GenerateNeighborhood automatically doesn't read in 19 4.
```

Figure 2: Structure of a topology file

```
InsertSimpleData(noOfbits,zipfExponent,p2p)
Input  : The length of each token in terms of number of bits
         The value of Zipf's exponent
Output : Peers with data.
```

The realistic data formation is accomplished by the following module:

```
InsertComplexData(noOfKeywords,nodeKeyword,zipfExponentp2p)
Input  : The number of unique keywords
         The average number of keywords at each node
         The value of Zipf's exponent
Output : Peers with data.
```

3. Building query list

The number of queries depends upon the number of times a particular experiment is intended to be repeated. Since the tokens of the query maintain Zipf's distribution, it is necessary to prepare the complete query list before the start of the experiment. Similar to the data distribution, here we also have to generate simple as well as realistic query lists. The query lists are generated by the following function.

```
GenerateQueryList(numberOfsearch,type,zipfExponent,queryOutput)
Input  : Number of times an experiment is repeated
         Simple or realistic
         The value of Zipf's exponent
Output : The total query list.
```

4. Setting update sequence

As mentioned above, each single node operates once in one time step, with the order of node updates following a random sequence. The initial random sequence is generated to start the network by the following function. We start with an array which has the node numbers written in order. The function *Shuffle* randomly selects n pairs of data and swaps them; n is the number of nodes. Therefore,

```
Shuffle(RandomArray)
Input : RandomArray
Output: RandomArray shuffled.
```

Once the peer-to-peer model is set up, the experiments are then performed. The experimental module is elaborated next.

1.2 Experiments

We perform two types of experiments - coverage and time-step. The basic structure of the two experiments is similar, only with varying exit conditions. The structure is elaborated below.

```
Experiment
{
InitializeSearch
for (each time-step; until exit condition not fulfilled)
          {
          Run the p2p network for one time step
          Collect related information
          }
}
```

The experiments are repeated for a specified number of times; we collect the information averaged over several runs. We next discuss the repetition number, exit conditions for each algorithm, and output derived from each experiment.

Coverage experiment

The coverage experiment is repeated $\mathcal{K}$ times. Each experiment runs until all the peers are not visited by the message packets. The output information consists of

1. Percentage of network covered.
2. Number of time steps.
3. Number of search items found.
4. Number of messages produced.

5. Number of operations performed. /* No of messages produced and no. of operations performed differ in any restricted version as we consider no of operations = no of messages produced + no of lookaheads performed to check whether a peer has already been visited or not. */

An example showing the output is as follows.

```
Input File Name = ../input/Rand.inp
RPM
Percent -  Turn Search Messages Operations
 21   -    19   155396    77     111
 32   -    21   231442   104     167
 43   -    23   307027   131     235
 53   -    24   381431   156     316
 63   -    25   454515   182     418
 73   -    26   526942   208     560
 83   -    28   592819   236     760
 91   -    29   657841   271   1128
100   -    41   716397   462   5579
```

The output information is the result derived from averaging the results of $\mathcal{K}$ runs.

The functions describing the coverage experiments are noted one by one. The *ExptRepeatCoverage* function performs repetition of *ExptCoverage* $\mathcal{K}$ times.

```
ExptRepeatCoverage(p2p,K,algorithm,message,
                   nodeOutput,queryOutput,exptOutput)
{
Input : The Network
        Number of times search has to be repeated
        Algorithm which has to be run
        Number of messages to be used
        Sequence of the nodes in which the search has to be initiated
        The exact query in those nodes.
Output: The output is stored in the array exptOutput. /*The output
        comprises of the fields described above.*/

for(i = 1 to K)
        {
        ExptCoverage(p2p,algorithm,message,node,query,exptResult)
        exptOutput = Avg(exptOutput,exptResult)
        }
}
```

The *ExptRepeatCoverage* calls the function *ExptCoverage,* where *ExptCoverage* is described below.

```
ExptCoverage(p2p,algorithm,message,node,query,exptResult)
```

```
{
Input :  The Network
         Algorithm which has to be run
         Number of messages to be used
         Initializing node
         Query to be processed
Output : The output is stored in the array exptOutput
         It has the same format as exptOutput.

Initialize(p2p,message,node,query,algorithm)
While(all nodes not visited by message packets)
      RunP2p(p2p,algorithm,exptResult)
}
```

The experiment *time-step* is described next.

Time-step experiment

The experiment is repeated for (say) $\mathcal{P}$ generations, where one generation is defined as a sequence of $\mathcal{Q}$ searches. That is, essentially the search is repeated $\mathcal{P} \times \mathcal{Q}$ times. Each experiment runs for a pre-defined number ($\mathcal{N}$) of time steps. We obtain output for each generation; this is of the following form.

1. Generation number.
2. Number of search items found.
3. Number of messages used.

An example showing the output is the following:

```
Input File Name = ../input/Rand1.inp
RPM
Gn. No. SearchItems  MessageUsed
   1   88.22          73.84
   2  116.66          85.38
   3   55.15         58.59
   4  144.93          96.54
   5   70.15          61.18
   6   71.56          68.76
   7  144.79         101.62
   8  156.57         105.19
   9   58.68          62.05
  10   89.62         69.66
```

The *ExptRepeatTimeStep* function takes care of the repetition of *ExptTimeStep* for $\mathcal{P} \times \mathcal{Q}$ number of times.

```
ExptRepeatTimeStep(p2p,P,Q,N,message,algorithm,nodeOutput,
                   queryOutput,exptOutput)
{
Input :  The Network
         Number of generations search has to be repeated
         Number of searches comprising a generation
         Number of time steps to be run
         Number of messages to be used
         Algorithm which has to be run
         Sequence of the nodes in which search has to be initiated
         Exact query in those nodes
Output : The output is stored in the 2-dimensional array exptOutput.
         Each row of exptOutput consists of the three fields
         - generation no., no. of search items, no. of messages used.
```

```
for(j = 1 to  P)
{
         for(i = 1 to Q)
         {
         ExptTimeStep(p2p,N,algorithm,node,query,exptResult)
         exptOutput[i] = Avg(exptOutput[i],exptResult)
         }
}
}
```

The *ExptRepeatTimeStep* calls the function *ExptTimeStep,* where *ExptTimeStep* is described below.

```
ExptTimeStep(p2p,N,algorithm,message,node,query,exptResult)
{
Input : The Network
        Number of time steps to be run
        Algorithm which has to be run
        Number of messages to be used
        Initializing node
        Query to be processed
Output :The output is stored in the array exptResult.

Initialize(p2p,message,node,query,algorithm)
for(N number of time steps)
        RunP2p(p2p,algorithm,exptResult)
}
```

We now elaborate the functions performed at each time step.

1.3 Running the p2p network at each time step

At each time step, all the nodes in the network check for incoming messages and perform actions upon them. The actions performed by them can be divided into three steps - (a) checking similarity between message and data; (b) forwarding the message; and (c) topology evolution. The sequence in which the nodes perform is random. The function *RunP2p* is noted below.

```
RunP2p(p2p,algorithm,exptResult)
{
Input :  The Network
         Algorithm which has to be run
Output : The output is stored in the array exptResult.

Shuffle(RandomArray)
for(each node)
```

```
      {
      k = CheckSimilarity
      N_new = DetemineNoMessage
      FowardMessage
      }
}
```

Each of the three functions are next explained one by one.

1. Similarity checking

The similarity measure differs for the two types of data distribution we have taken into consideration. So CheckSimilaritySimple returns either 0 or 1, while CheckSimilarityReal returns the actual number of keywords matched. We note the input/output of the functions below.

```
k = CheckSimilaritySimple(node.message, node.content, noOfSimilarBits)
Input :  The first message in the node's message queue
         The node's content
Output : No. of bits similar between node.message and node.content.
         /*this output is required as an input for the next function*/
         k - indicating similarity (0/1)
```

For the realistic data type, the function is:

```
k = CheckSimilarityRealistic(node.message,node.content)
Input :  The first message in the node's message queue
         The node's content
Output : k - Number of keywords found similar.
```

2. Determining the number of packets to be forwarded

The number of message packets which will be forwarded to the neighborhood depends upon the types of algorithms - random walk, or affinity-driven proliferation-mutation - we run.

```
int DetermineNoMessage(algorithm,similarity measures)
{
return N_new
}
```

Once the number N_{new} of new packets is determined, the packets are forwarded through the function described next.

3. Forwarding the message packets

Forwarding the message packets broadly falls under two categories - restricted movement, and simple or unrestricted movement. The message forward function is noted.

```
ForwardMessage(node,N_new,category,p2p)
Input : The node which forwards the message
        Number of messages to be forwarded
        The type of forwarding scheme
        (restricted, unrestricted, high-degree etc.)
Output : The forwarded message in p2p network.
{
If (category = 1)
        MessageForwardRestricted(node,N_new,category,p2p)
If (category = 2)
        MessageForwardUnrestricted(node,N_new,category,p2p)
}
```

This concludes our description of the implementation of immune-based search algorithm. We now discuss the input function.

1.4 Input

The input consists of various parameters which we note one by one.

1. Topology file - The file which contains the information on the topology of the peer to peer network. Fig. 2 shows the details of the file.
2. Number of nodes
3. Action Type
   - <TimeStep> / <Coverage>; <How Many Times> <Generation Length>
4. Type of data/query distribution
   - <Simple> / <Realistic>; <no. of bits> / <no of keywords>
5. <Proliferation constant> <Mutation constant>
6. <Algorithms to be performed>

We elaborate the input file configuration with the following example file. The lines starting with '#' are lines which are not in the input file, but are used here only for the purpose of explanation.

```
inet.txt
#a powerlaw graph as input.

10000
#no. of nodes

1 1000 0
#1 - indicates experiment coverage;
#1000 - indicates no.of times it is repeated;
#0 - is a dummy parameter, it is there to maintain equivalence
#with time-step experiment which requires two parameters.

1 2000
#data type - realistic and no of keywords used - 2000

0.5 0.0
#<proliferation constant>; <mutation constant>
# For this version mutation constant = 0

1 0 1 0
# <RPM  executed> <PM not executed> <RW executed> <RRW not executed>
```

1.5 File arrangement

The files are arranged in several directories. We specify the directories and the files inside the directories one by one.

- immuneSearch/hfile
  - initial_population.h - functions to build the topology.
  - Information.h - functions to build the contents of the network and query set.
  - Movement.h - functions to guide the movement at each generation.
  - Experiment.h - functions defining the experiments.
  - small_function.h - miscellaneous functions.
  - header.h - information regarding various variables.
- immuneSearch/bin
  - immune.c - the main file to run the program.
  - makefile.
- immuneSearch/graphs - files containing topology information of different graphs. e.g. power_10000 - a powerlaw graph with 10000 nodes; rand10000 - an uniform random graph with 10000 nodes; Grid.100,00 - a grid with 10000 nodes; Gnutella1095 - a graph representing realistic Gnutella topology comprising of 1095 nodes.

- immuneSearch/input - files containing different possible input files. The results of the sample input files are stored in immuneSearch/bin/RESULT

Besides these files, there is a file grid.c which generates the grid topology in the directory TopologyGenerator/GRID.

1.6 Running and compiling the files

Compiling the main file immune.c can be done just by running the make file. Similarly, the files in topology generators - convert.c and grid.c - can be compiled with the following respective commands.

```
gcc grid.c -lm -o grid.out
```

Running the main program is done from the directory immuneSearch/bin with the command

```
immunesearch.out inputfilename
```

The other executable file grid.out is run with the following commands, respectively:

```
grid.out outputfilename NumberOfNodes length breadth
```

References

[1] C Jin, Q Chen, and S Jamin. Inet: Internet Topology Generator. University of Michigan Technical Report CSE-TR-433-00, 2002.

[2] A Medina, A Lakhina, I Matta, and J Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS*, August 2001.