Source Terminal Network Unreliability Estimation by Crude Monte Carlo (1)

v1.1

Leslie Murray
Universidad Nacional de Rosario

An undirected graph $G = (V, E)$, where V is a set of n absolutely reliable nodes and E a set of m independent unreliable links, is a frequently used model for studying communication network reliability. Being $X_i = 1$ when the i^{th} link is *operative* and $X_i = 0$ when the i^{th} link is *failed*, vector $\mathbf{X} = (X_1, X_2, \cdots, X_m)$ denotes the state of the links and, thereby, the state of the whole network. The single link reliability is defined as $r_i = P[X_i = 1]$ whereas the single link unreliability as $q_i = 1 - r_i = P[X_i = 0]$.

$\Phi(\mathbf{X})$ is called structure function, a function that equals 1 when the network is *operative* and 0 when the network is *failed*. In this program $\Phi(\mathbf{X})$ is associated to the source–terminal network reliability model in which the network is considered *operative* if there is a path of *operative* links between two nodes s and t, and *failed* if there is no path of *operative* links between nodes s and t. By means of this function, the network reliability R and unreliability Q are defined, respectively, as $R = P[\Phi(\mathbf{X}) = 1]$ and $Q = P[\Phi(\mathbf{X}) = 0]$.

Straightforward (also called Standard, Direct or Crude) Monte Carlo estimations of the network reliability and unreliability can be computed, respectively, as $\widehat{R} = 1/N \sum_{i=1}^{N} \Phi(\mathbf{X}^{(i)})$ and $\widehat{Q} = 1/N \sum_{i=1}^{N} (1 - \Phi(\mathbf{X}^{(i)})$ where $\mathbf{X}^{(i)}$, $i = 1, \cdots, N$ are *i.i.d.* samples taken from $f_{\mathbf{X}}(\mathbf{x})$: the probability mass function of vector $\mathbf{X}$. For highly reliable networks most of the $X^{(i)}$ samples will equal 1 and, as a consequence, most of the replications $\Phi(\mathbf{X}^{(i)})$ will equal 1 also. For extremely reliable networks, with unreliabilities in the order of 10^{-10} or even less, it will take an average of 10^{10} replications to get $\Phi(\mathbf{X}^{(i)}) = 0$ at least once. For these networks $\Phi(\mathbf{X}) = 0$ is a rare event.

The accuracy of the unreliability estimation can be assessed by means of a relative error defined as $V\{\widehat{Q}\}^{1/2}/E\{\widehat{Q}\}$, that in the case of Crude Monte Carlo takes the form $((1 - Q)/NQ)^{1/2} \approx (1/NQ)^{1/2}$. This shows a weakness of the method for the case of highly reliable networks (Q very low) and the reason why accurate estimations require a high number of replications (N very high).

As for the program $E\{\widehat{Q}\}$ and $V\{\widehat{Q}\}$ are both unknown, the following unmbiased estimators are calculated, instead:

$$1/N \sum_{i=1}^{N} (1 - \Phi(\mathbf{X}^{(i)}) \qquad \text{for} \qquad E\{\widehat{Q}\}$$

$$\sum_{i=1}^{N} (1 - \Phi(\mathbf{X}^{(i)})^2/(N(N-1)) - \left(\sum_{i=1}^{N} (1 - \Phi(\mathbf{X}^{(i)})\right)^2/(N-1) \qquad \text{for} \qquad V\{\widehat{Q}\}$$

This program attempts to obtain these estimators and, by means of them, the relative error too. The network under study is passed to the program as a text file with the following format:

<**node** s>
<**node** t>
<n>
<l>

<**node** 1>	$< adj_1 >$	<**node** dest>	<*rlb*>	<**node** dest>	<*rlb*>	$\cdots$
<**node** 2>	$< adj_2 >$	<**node** dest>	<*rlb*>	<**node** dest>	<*rlb*>	$\cdots$
<**node** 3>	$< adj_3 >$	<**node** dest>	<*rlb*>	<**node** dest>	<*rlb*>	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
<**node** n>	$< adj_n >$	<**node** dest>	<*rlb*>	<**node** dest>	<*rlb*>	$\cdots$

All expressions like $< \ldots >$, are integers numbers. <**node** s> and <**node** t> are, respectively, the *source* and *terminal* nodes (to estimate the source–terminal unreliability). <n> is the number of *nodes* and <l> the number of *links*. Every one of the subsequent lines have the following meaning: <**node** i > has a number $< adj_i >$ of adjacent nodes, each one of them identified by the correponding number (<**node** dest>) followed by its single reliability value (<*rlb*>).

February 11, 2009 at 00:42

1. The program general structure. The whole program is shown as a set of sections, each one of them is introduced in the following items.

```
⟨ Library Headers and Included Files 2 ⟩;
⟨ New Type Definition and Global Variables 4 ⟩;
⟨ Prototypes of Auxiliary Functions 5 ⟩;
int main(int argc, char *argv[])
{
  ⟨ Local Variables of main() 6 ⟩;
  ⟨ Input Data Validation 7 ⟩;
  ⟨ Crude Monte Carlo Algorithm 8 ⟩;
  ⟨ Print of Output 9 ⟩;
  return 0;
}
⟨ Auxiliary Functions 10 ⟩;
```

2. At the top, the inclusions are as usual, mostly to allow the use of input-output and mathematical functions. The most remarkable files are the library `time.h` to make use of functions to measure the execution time and `mt19937ar.c`, the code of the Mersenne Twister random numbers generator.

```
⟨ Library Headers and Included Files 2 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mt19937ar.c"
```

This code is used in section 1.

3. For clarification, single random numbers from the Mersenne Twister are called U.

```
#define U genrand_real1()   /* a single random number (from the Mersenne Twister) */
```

4. Some structures are to be allocated: one unit of **struct link** for every link and one unit of **struct network** for the whole network. In the input file from which the network topolgy is loaded, the nodes are enumerated from 1 to *numnodes*. The adjacency list of the graph (network) makes use of these numbers. Such adjacency lists are rebuilt every time a new state is sampled for every link, and the latest version of the adjacency lists is used by the algorithms to determine whether nodes *s* and *t* are connected.

⟨ New Type Definition and Global Variables 4 ⟩ ≡

```
typedef struct link {
  int adj;   /* 1 or 0 to build the adjacency martix of the network */
  double rlb;   /* single link reliability */
  int smp;   /* 1 or 0 to build the adjacency matrix of the network, after random fail */
} lnk, *pt_lnk;
typedef struct network {
  int s;   /* source node */
  int t;   /* target node */
  int numnodes;   /* number of nodes */
  int numlinks;   /* number of links */
  struct link **I;   /* matrix of links, size: I[numnodes+1][numnodes+1] (I[0][0], unused) */
} net, *pt_net;
int **list;   /* matrix to perform as "list" of "adjacency lists" */
int *visited;   /* array to keep track of the visited nodes in the DFS */
int connected;   /* 1 if there is a path connecting "s" and "t" and 0 otherwise */
int seed;   /* seed for the random numbers generator Mersenne Twister */
```

This code is used in section 1.

5. All functions and algorithms deal with a network whose topolgy is passed to the program as the argument *argv*[1] of the *main*( ). This argument is the name of a file that is finally passed to the function *Initialize* as the argument *filename*. Function *Initialize* allocates one unit of a **struct network** and returns a pointer to it. This pointer is taken as the argument by all the other functions, namely: *X*( ), *Fail*( ), `DFS`( ) and *Phi*( ). All these functions are thought to operate this way: *X*( ) sets a random state on every link by means of a numerical value, if 0, the link is removed from the network, if 1, the link remains untouched; *Fail*( ) sets a random state on every link; `DFS`( ) performs a Depth First Search starting from node *s* and tells *Phi*( ) whether node *t* is reached or not, if node *t* is reached *Phi*( ) returns 1, otherwise it returns 0.

⟨ Prototypes of Auxiliary Functions 5 ⟩ ≡

```
pt_net Initialize(char *filename);   /* allocates a network associated to the info in file filename */
int X(int node1, int node2, pt_net nt);   /* returns 1 if link node1−node2 is operative, 0 otherwise */
void Fail(pt_net nt);   /* set link random fails and builds the adjacency lists after that fail */
void DFS(int node, pt_net nt);   /* performs DFS from node s, stops when node t is reached */
int Phi(pt_net nt);   /* returns 1 if s and t are connected and 0 otherwise */
```

This code is used in section 1.

6. Function *main*( ) makes use of many local variables, none of which deserve a particular explanation. Most of them operate as indexes and as memory units to retain some values during mathematical calculation. *n* is a pointer to the network under study. *ti* receives the value of function *clock*( ) (library `time.h`). Function *clock*( ) returns the elapsed time since the beginning of the execution.

⟨ Local Variables of *main*( ) 6 ⟩ ≡

```
int i, j, k, S, size;
double X, V, Q, t;
pt_net n;
clock_t ti;
```

This code is used in section 1.

7. The program accepts input data at the command line. If compilation is such that `crude1` is the name of the executable, the program runs as:

```
./crude1 <File> <Size> <Seed>
```

where `<File>` is the text file with the network topolgy info, `<Size>` the number of Monte Carlo trials and `<Seed>` the value to set the starting point of the random numbers generator (Mersenne Twister). Some validation on these data is aimed to check: the number of inputs to be not less, and not more than three, the value of `<Size>` expecting that is not less than one and the `<Seed>`, restricting it to positive values.

⟨ Input Data Validation 7 ⟩ ≡

```
if (argc < 4) {
  printf("\n␣Some␣input␣data␣is␣missing!␣Run␣as:\n");
  printf("\n␣./executable␣<File>␣<Size>␣<Seed>\n\n");
  exit(1);
}
if (argc > 4) {
  printf("\n␣You've␣entered␣more␣data␣than␣necessary!␣Run␣as:\n\n");
  printf("\n␣./executable␣<File>␣<Size>␣<Seed>\n\n");
  exit(1);
}
if ((size = atoi(argv[2])) < 1) {
  printf("\n␣The␣number␣of␣trials␣can␣not␣be␣less␣than␣1!␣Run␣as:\n");
  printf("\n␣./executable␣<File>␣<Size>␣<Seed>\n\n");
  exit(1);
}
if ((seed = atoi(argv[3])) < 0) {
  printf("\n␣The␣seed␣can␣not␣be␣negative!␣Run␣as:\n");
  printf("\n␣./executable␣<File>␣<Size>␣<Seed>\n\n");
  exit(1);
}
```

This code is used in section 1.

8. As explained in the introduction, the core of this program is a very simple algorithm, aimed to repeat a number *size* of times a cycle in which function *Fail*( ) sets a random value of either 0 or 1 to every link; after this, function *Phi*( ) returns 1 if nodes s and t are connected and 0 otherwise. Accumulation of the complement of the value of *Phi*( ), and its square, let the estimation of the unreliability Q and its corresponding variance V be done. *ti* saves the value of the function *clock*( ) just before the algorithm starts, and after completion of the run *ti* is subtracted from the current value of *clock*( ). Such difference is the execution time of the algorithm (note that the initialization process time is not considered).

⟨ Crude Monte Carlo Algorithm 8 ⟩ ≡

```
n = Initialize(argv[1]);
ti = clock();
S = 0;
X = 0.0;
V = 0.0;
for (k = 0; k < size; k++) {
   Fail(n);
   X = (1 − Phi(n));
   S += X;
   V += X ∗ X;
}
Q = (double) S/size;
V = (V/size − Q ∗ Q)/(size − 1);
t = (double)(clock() − ti)/CLOCKS_PER_SEC;
```

This code is used in section 1.

9. The reason why this version is called "(1)" will be revealed with the advent of versions (2) and (3)... At the moment it's enough to say that this is the simplest version (at least the simplest out of the three) to implement the basis of the Crude Monte Carlo algorithm for the estimation of network unreliability, Q. The main output is therefore the value of Q. This value is shown toghether with the name of the network under study, the number of Monte Carlo replications, the execution time in seconds, the variance V, the standard deviation $V^{1/2}$ and the relative error $V^{1/2}/Q$.

⟨ Print of Output 9 ⟩ ≡

```
printf("\n␣␣Network:␣%s␣␣␣Replications:␣%d␣␣␣ExecTime=%f", argv[1], size, t);
printf("\n␣␣*******************␣CRUDE␣MONTE␣CARLO␣(1)␣******************");
printf("\n␣␣␣␣␣␣␣Unreliability␣␣␣␣Q␣=␣%1.16f␣=␣%1.2e", Q, Q);
printf("\n␣␣␣␣␣␣␣Variance␣␣␣␣␣␣␣␣␣V␣=␣%1.16f␣=␣%1.2e", V, V);
printf("\n␣␣␣␣␣␣␣Std.␣Dev.␣␣␣␣␣␣␣SD␣=␣%1.16f␣=␣%1.2e", sqrt(V), sqrt(V));
printf("\n␣␣␣␣␣␣␣Relative␣Error␣␣RE␣=␣%1.16f␣=␣%1.2f%%", sqrt(V)/Q, 100 ∗ sqrt(V)/Q);
printf("\n␣␣************************************************************\n\n");
```

This code is used in section 1.

10. The set of Auxiliary Functions. These functions provide support to implement the main operations required by the program. They are shown here, classified in three sections, each one of them containing code clearly associated to the name given to it.

$\langle$ Auxiliary Functions 10 $\rangle \equiv$
  $\langle$ Initialization 11 $\rangle$;
  $\langle$ Fail Generation 12 $\rangle$;
  $\langle$ Function of Structure 13 $\rangle$;

This code is used in section 1.

11. Function *Initialize*( ) builds up and allocates the network data structure **net** with information read from the file that holds the network data (*filename*). It also initializes the random numbers generator Mersenne Twister.

⟨ Initialization 11 ⟩ ≡

```
pt_net Initialize(char *filename)
{
  pt_net pt_n;
  FILE *fp;
  int i, j, node1, node2, num;
  double reliability;
      /* Allocate one unit of the structure net to hold the network info */
  if ((pt_n = (pt_net) calloc(1, sizeof(net))) ≡ Λ) {
    printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
    exit(1);
  }
        /* Open the file with the network info and scan for the source and target */
        /* node, the number of nodes and the number of links at the top of it */
  if ((fp = fopen(filename, "r")) ≡ Λ) {
    printf("\n␣Fail␣attempting␣to␣open␣a␣disk␣file...\n");
    exit(1);
  }
  fscanf(fp, "%d", &pt_n→s);
  fscanf(fp, "%d", &pt_n→t);
  fscanf(fp, "%d", &pt_n→numnodes);
  fscanf(fp, "%d", &pt_n→numlinks);
    /* Allocate matrix I with a size of (numnodes+1)*(numnodes+1), and link it to */
    /* the corresponding pointer of network net */
  if ((pt_n→I = (pt_lnk *) calloc(pt_n→numnodes + 1, sizeof(struct link *))) ≡ Λ) {
    printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
    exit(1);
  }
  for (i = 0; i ≤ pt_n→numnodes; i++) {
    if ((pt_n→I[i] = (pt_lnk) calloc(pt_n→numnodes + 1, sizeof(struct link))) ≡ Λ) {
      printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
      exit(1);
    }
  }
        /* Initialize matrix I with 0s for the adjacencies and 0.0s for the */
        /* reliabilities of every link */
  for (i = 1; i ≤ pt_n→numnodes; i++)
    for (j = 1; j ≤ pt_n→numnodes; j++) {
      pt_n→I[i][j].adj = 0;
      pt_n→I[i][j].rlb = 0.0;
      pt_n→I[i][j].smp = 0;
    }
          /* Load matrix I values from the file with the network info and set the */
          /* corresponding values of adjacency and reliability */
  for (i = 1; i ≤ pt_n→numnodes; i++) {
    fscanf(fp, "%d%d", &node1, &num);
    for (j = 1; j ≤ num; j++) {
      fscanf(fp, "%d%lf", &node2, &reliability);
      pt_n→I[node1][node2].adj = 1;
```

```
      pt_n→I[node1][node2].rlb = reliability;
    }
  }
  fclose(fp);
      /* Take the seed value and initialize the pseudorandom numbers generator */
      /* Mersenne Twister */
  init_genrand(seed);
      /* Allocate matrix list of size (numnodes+1)*(numnodes+1), same as matrix I */
  if ((list = (int **) calloc(pt_n→numnodes + 1, sizeof(int *))) ≡ Λ) {
    printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
    exit(1);
  }
  for (i = 0; i ≤ pt_n→numnodes; i++) {
    if ((list[i] = (int *) calloc(pt_n→numnodes, sizeof(int))) ≡ Λ) {
      printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
      exit(1);
    }
  }
        /* Initialize matrix list, filling it with 0s */
  for (i = 1; i ≤ pt_n→numnodes; i++)
    for (j = 1; j ≤ pt_n→numnodes; j++) {
      list[i][j] = 0;
    }
          /* Allocate the visited array and set 0 for all nodes (not visited) */
  if ((visited = (int *) calloc(pt_n→numnodes, sizeof(int))) ≡ Λ) {
    printf("\n␣Fail␣attempting␣to␣allocate␣memory...\n");
    exit(1);
  }
  for (i = 1; i ≤ pt_n→numnodes; i++) visited[i] = 0;
  connected = 0;
  return pt_n;
}
```

This code is used in section 10.

12. Depending on the probability distribution of the link between *node*1 and *node*2, function $X()$ returns 1 if such link is operative and 0 otherwise. It is accepted that function $X()$ is only required for links that actuallly exist, i.e. links for which $I[node1][node2].adj = 1$.

Function *Fail*( ) sets a random value of 0 or 1 in *smp* for every existing link of network *nt*. This way, a randomly failed network is built up and saved into matrix $I[\,][\,].smp$ and also in $list[\,][\,]$. It is accepeted that the network graph is undirected and that the reliability of every link has the same value in both directions. It is therefore unnecesary to sample all the matrix elements, it suffices to do it for all the elements above the diagonal and to set the same value to the simetric element (the diagonal elements are all 0).

⟨ Fail Generation 12 ⟩ ≡

```
int X(int node1, int node2, pt_net nt)
{
  if (U < nt→I[node1][node2].rlb) return 1;
  else return 0;
}
void Fail(pt_net nt)
{
  int i, j, k;
  for (i = 1; i ≤ nt→numnodes; i++)
    for (j = i + 1; j ≤ nt→numnodes; j++)
      if (nt→I[i][j].adj) {
        nt→I[i][j].smp = (nt→I[i][j].adj ∧ X(i, j, nt));
        nt→I[j][i].smp = nt→I[i][j].smp;
      }
  for (i = 1; i ≤ nt→numnodes; i++) {      /* Clean the prior adjacency lists */
    k = 1;
    while (list[i][k] > 0) {
      list[i][k++] = 0;
    }
  }
  for (i = 1; i ≤ nt→numnodes; i++) {      /* Create the new adjacency lists */
    k = 1;
    for (j = 1; j ≤ nt→numnodes; j++) {
      if (nt→I[i][j].smp) {
        list[i][k++] = j;
      }
    }
  }
  return;
}
```

This code is used in section 10.

13. Functions *Phi*( ) and `DFS`( ) both update the value of variable *connected*. *Phi*( ) sets it to 0 (as an indication that there is no path of operative links between node s and t). It also assumes that no node has been visited by the Depth First Search yet. Then it calls `DFS`( ) that, starting frome node s sets *connected* to 1 and stop if there is a path of operative links between nodes s and t, and does nothing if there is not such path.

⟨ Function of Structure 13 ⟩ ≡

```
int Phi(pt_net nt)
{
  int k;
  connected = 0;
  for (k = 1; k ≤ nt→numnodes; k++) visited[k] = 0;
        /* DFS() makes a Depth First Search from node s and: */
        /* - returns 1 if node t is reached */
        /* - returns 0 if node t is not reached */
  DFS(nt→s, nt);
  return connected;
}
void DFS(int node, pt_net nt)
{
  int k;
  if (connected) return;
  if (node ≡ nt→t) {
    connected = 1;
    return;
  }
  visited[node] = 1;
  k = 1;
  while (list[node][k] > 0) {
    if (¬visited[list[node][k]]) DFS(list[node][k], nt);
    k++;
  }
  return;
}
```

This code is used in section 10.

14. Index.

CRUDE1