Application of network reliability using exhaustive enumeration - Project 2

Harshavardhan Nalajala

July 10 2017

Contents

1 Description

2 Algorithm

3 Input

```
input.txt:
5
10
1
1 2
1 3
```

```
1: procedure calculateReliability
2:     for i in numEdges do
3:         list ← getCombinations(edges, i)
4:         for each combination in list
5:         edgeMatrix ← constructEdgeMatrix(combination)
6:         if connectedUsingBFS(edgeMatrix) then
7:             reliability ← reliability + probability(combination)
       return reliability
```

```
1: procedure constructEdgeMatrix(combination)
2:     for i in numNodes do
3:         if combination[i] == 0 then
4:             s ← edge[i][0]
5:             d ← edge[i][1]
6:             edgeMatrix[s][d] = 1
7:             edgeMatrix[d][s] = 1
       return edgeMatrix
```

```
1: procedure getCombinations(startindex, endindex, i)
2:     if startindex = endindex then
3:         if k = 0 then return "0"
           return "1"
4:     list = getCombinations(startindex + 1, endindex, i)
5:     for list in list do
6:         nlist ← "0" + list
7:     list = getCombinations(startindex + 1, endindex, i − 1)
8:     for list in list do
9:         nlist ← "1" + list
       return nlist
```

```
1: procedure connectedUsingBFS(edgeMatrix)
2:     for i in numNodes do
3:         vertex[i] ← 0
4:     add 0 to queue
5:     while queue is not empty do
6:         for j in numNodes do
7:             if edgeMatrix[i][j] != 0 and vertex[i] != 1 then
8:                 insert j to queue
9:                 vertex[j] ← 1
10:    for j in vertex do
11:        if  thenvertex[j] == 0 return 0
       return 1
```

```
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
0.05
```

4 Output

```
Probability     Reliability
0.050000        0.000569
0.100000        0.005009
0.150000        0.018029
0.200000        0.044373
0.250000        0.087816
0.300000        0.150322
0.350000        0.231491
0.400000        0.328416
0.450000        0.435968
0.500000        0.547485
0.550000        0.655738
0.600000        0.753997
0.650000        0.837022
0.700000        0.901767
0.750000        0.947684
0.800000        0.976563
0.850000        0.991951
0.900000        0.998283
0.950000        0.999884
1.000000        1.000000
// 1024 combinations and total reliability of network with p = 0.9
Total combinations: 1024 Reliability: 0.998283
// Run k from 0 to 20, reliability values
0.998283
0.997346
0.996311
0.995507
```

```
0.995024
0.992530
0.991743
0.991043
0.990579
0.990784
0.989484
0.987036
0.987080
0.985806
0.983896
0.984593
0.982788
0.980476
0.980215
0.982170
0.979959
```

5 Analysis

As reliability of each link increases, overall reliability of the whole system increases since probability of each link in up state is directly proportional to overall system up state. This is depicted in graph of figure 1 at the end.

728 combinations belong to system up state and 295 combinations belong to system down state. Choosing a combination randomly is biased towards system up state and reversing the state will reduce the reliability. So, with increasing k, reliability of the whole system reduces gradually except one case where reliability of the system dips down with k = 19. Gradual reduction in reliability is explained based on system up random combinations chosen are greater than system down combinations. Sudden dip happened since random combinations are all system up combinations.

6 ReadMe

Run Makefile using following command in the directory.

```
make -f Makefile
```

Makefile generates the executabla(reliability). Executable expects input in the following format. Run command: ./reliability ¡ input.txt

- number of nodes, n

- number of edges, N
- N lines of two space separated integers i, j denoting the edge between i and j
- initial probability, p

7 Appendix

Files:

- graph utils.h, graph utils.c: API to get the edge matrix from given combination and API implementation of BFS to check network connectivity.
- combinations.h combinations.c: API to create all combinations of 0s and 1s, API to print and free the list of combinations. Linked list used for dynamic memory management.
- reliability.c: Main file containing API to calculate reliability of each edge, reliability of overall system using combinations API and graph utils API.

```
combination.h:
#ifndef __COMBINATIONS_H
#define __COMBINATIONS_H

typedef struct list {
        char *c;
        int len;
        struct list *next;
}llist_st;

extern llist_st *getCombinations(int start, int end, int k);
extern void free_list(llist_st *);
extern void print(llist_st *);
extern llist_st *reverseList(llist_st *);
#endif

#ifndef _GRAPH_UTILS_H
#define _GRAPH_UTILS_H
```

```
/* Construct graph using edges and character array depicting links up/down
void graph_utils_construct(int numNodes, int edgeMatrix[][numNodes], char
/* Check if the graph given by edge matrix is disconnected i.e. is any node
int graph_utils_disconnected(int numNodes, int edgeMatrix[][numNodes]);

typedef struct queue {
        int list[100];
        int front;
        int back;
}queue_st;

#endif
```

graph_utils.c:

```
#include <stdio.h>
#include "graph_utils.h"

void initialize(int numNodes, int edgeMatrix[][numNodes]) {
        int i = 0;
        int j = 0;
        for (i = 0; i < numNodes; i++) {
                for (j = 0; j < numNodes; j++) {
                        edgeMatrix[i][j] = 0;
                }
        }
}

/* Construct graph using edges and character array depicting links up/down
void graph_utils_construct(int numNodes, int edgeMatrix[][numNodes], char
        initialize(numNodes, edgeMatrix);
        int i = 0;
        for (i = 0; i < numEdges; i++) {
                int s = edges[i][0];
                int d = edges[i][1];
                if (c[i] == '0') {
                        // link is not down
                        // assuming undirected graph
                        edgeMatrix[s][d] = 1;
                        edgeMatrix[d][s] = 1;
```

```
                }
        }
}

void queue_insert(queue_st *q, int val) {
        if (q->back == (q->front+1)%100) {
        } else {
                q->list[q->front] = val;
                q->front += 1;
        }
}

int queue_remove(queue_st *q) {
        if (q->front == q->back) {
                return -1;
        } else {
                int val = q->list[q->back];
                q->back += 1;
                return val;
        }
}

/* Check if the graph given by edge matrix is disconnected i.e. is any node
int graph_utils_disconnected(int numNodes, int edgeMatrix[][numNodes]) {
        // assuming undirected graph
        int i = 0;
        // assuming undirected graph
        int i = 0;
        // run BFS
        int vertex[numNodes];
        for (i = 0; i < numNodes; i++) {
                vertex[i] = 0;
        }
        queue_st q;
        q.front = 0;
        q.back = 0;
        queue_insert(&q, 0);
        vertex[0] = 1;
        int val = queue_remove(&q);
        while (val != -1) {
```

```
                int j = 0;
                for (j = 0; j < numNodes; j++) {
                        if (edgeMatrix[val][j] != 0 && vertex[j] != 1) {
                                queue_insert(&q, j);
                                vertex[j] = 1;
                        }
                }
                val = queue_remove(&q);
        }

        for (i = 0; i < numNodes; i++) {
                if (vertex[i] == 0) {
                        return 1;
                }
        }
        return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "combinations.h"

char *add(char *first, char *second) {
        int len1 = strlen(first);
        int len2 = strlen(second);

        char *c = (char *)malloc(sizeof(char)*(len1+len2+1));
        memset(c, '\0', len1+len2+1);
        strncpy(c, first, len1);
        strncpy(c+len1, second, len2);
        return c;
}

llist_st *getCombinations(int start_index, int end_index, int k) {
        if (start_index > end_index) {
                return NULL;
        }

        if (start_index == end_index) {
```

```
        if (k == 0) {
                llist_st *list = (llist_st *)malloc(sizeof(llist_st
                list->c = "0";
                list->len = 1;
                list->next = NULL;
                return list;
        } else if (k == 1) {
                llist_st *list = (llist_st *)malloc(sizeof(llist_st
                list->c = "1";
                list->len = 1;
                list->next = NULL;
                return list;
        } else {
                return NULL;
        }
} else if (end_index - start_index +1 == k) {
        llist_st *list = (llist_st *)malloc(sizeof(llist_st));
        list->c = (char *)malloc(sizeof(char)*(k+1));
        int i = 0;
        for (i = 0; i < k; i++) {
                (list->c)[i] = '1';
        }
        (list->c)[k] = '\0';
        list->len = k;
        list->next = NULL;
        return list;
} else {
        llist_st *head = NULL;
        llist_st *temp_parent = NULL;
        llist_st *head = NULL;
        llist_st *temp_parent = NULL;
        llist_st * list = getCombinations(start_index+1, end_index
        while (list != NULL) {
                char *list_c = list->c;
                int len = list->len;
                llist_st *temp = (llist_st*)malloc(sizeof(llist_st
                temp->c = add("0", list_c);
                temp->len  = len+1;
                temp->next = NULL;
                if (head == NULL) {
```

```
                        head = temp;
                    } else if (temp_parent != NULL) {
                        temp_parent->next = temp;
                    }
                    temp_parent = temp;
                    list = list->next;
                }
                list = getCombinations(start_index+1, end_index, k-1);
                while (list != NULL) {
                    char *list_c = list->c;
                    int len = list->len;
                    llist_st *temp = (llist_st*)malloc(sizeof(llist_st
                    temp->c = add("1", list_c);
                    temp->len  = len+1;
                    temp->next = NULL;
                    if (head == NULL) {
                        head = temp;
                    } else if (temp_parent != NULL) {
                        temp_parent->next = temp;
                    }
                    temp_parent = temp;
                    list = list->next;
                }
                return head;
        }
}

int get(char *c) {
        int len = strlen(c);
        int i = 0;
        int ret = 0;
        while (i < len) {
                ret = 2*ret + (c[i]-'0');
                i++;
        }
        return ret;
}

void free_list(llist_st *list) {
        while (list != NULL) {
```

```
                llist_st *temp = list;
                list = temp->next;
                free(list);
        }
}

void print(llist_st *list) {
        while (list != NULL) {
                printf("%s, ", list->c);
                list = list->next;
        }
        printf("\n");
}

llist_st *reverseList(llist_st *list) {
        llist_st *start = list;
        llist_st *temp1 = list;
        llist_st *temp2 = list->next;
        while (temp1 && temp2) {
                llist_st *local = temp2->next;
                temp2->next = start;
                temp1->next = local;
                start = temp2;
                temp2 = local;
        }

        return start;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "combinations.h"
#include "graph_utils.h"

/* Function to calculate probability of each link */
void setReliabilityOfEdge(double *reliabilities, int numEdges, double p_ba
        int i = 0;
        double magic[] = {2,0,2,1,3,4,6,8,3,5};
```

```
        // get probability as p ^ ceil(magic[i] / 3)
        for (i = 0; i < numEdges; i++) {
                reliabilities[i] = pow(p_base,  (ceil(magic[i]/3)));
        }
}

/* Returns reliability of the link based on up/down state */
double getProbability(char *c, int num, double *reliabilities) {
        double prob = 1;
        int i = 0;
        while (i < num) {
                if (c[i] == '0') {
                        prob = prob *reliabilities[i];
                } else {
                        prob = prob *(1-reliabilities[i]);
                }
                i++;
        }
        return prob;
}

/* Returns 1 if graph of network is disconnected. Returns 0 otherwise */
int isSystemDown(int numNodes, char *c, int num, int edges[][2]) {
        int edgeMatrix[numNodes][numNodes];
        graph_utils_construct(numNodes, edgeMatrix, c, num, edges);
        if (graph_utils_disconnected(numNodes, edgeMatrix) == 1) {
                return 1;
        } else {
                return 0;
        }
}

/* Reliability calculator using all possible combinations of links' states
double calculateReliability(int numNodes, int numEdges, double *reliabiliti
        int i = 0;
        double reliability = 0;

        for (i = 0; i < numEdges; i++) {
                // all combinations are returned as linked list
                // get combinations of i down links
```

```
                // all combinations are returned as linked list
                // get combinations of i down links
                // up --> 0
                // down --> 1
                llist_st *list = getCombinations(0, numEdges-1, i);
                llist_st *temp = list;
                while (temp != NULL) {
                        // for each combination, check if graph of the netv
                        if (isSystemDown(numNodes, temp->c, numEdges, edges
                                // if graph is not disconnected, add relia
                                reliability = reliability + getProbability
                        }
                        temp = temp->next;
                }
                free_list(list);
        }
        return reliability;
}

llist_st *getSequence(llist_st *list, int seq) {
        while (seq > 0 && list != NULL) {
                list = list->next;
                seq -= 1;
        }
        return list;
}

char *switchStates(char *c, int numEdges) {
        int i = 0;
        char *newC = (char *)malloc(sizeof(char)*(strlen(c)+1));
        newC[strlen(c)] = '\0';
        while (i < numEdges) {
                if (c[i] == '0') {
                        newC[i] = '1';
                } else {
                        newC[i] = '0';
                }
                i++;
        }
        return newC;
```

```
}

int main() {
        // get number of nodes
        int numNodes = 0;
        scanf("%d", &numNodes);

        // get number of edges
        int numEdges = 0;
        scanf("%d", &numEdges);

        // is graph undirected
        int isUndirected = 0;
        scanf("%d", &isUndirected);

        int i = 0;
        // Edge list
        int edges[numEdges][2];
        for (i = 0; i < numEdges; i++) {
                int s = 0;
                int d = 0;
                scanf("%d %d", &s, &d);
                edges[i][0] = s-1;
                edges[i][1] = d-1;
        }

        double p_base = 0;
        scanf("%lf", &p_base);
        double p = p_base;
        printf("Probability \t Reliability\n");
        // case 1: probability of each link run from 0.05 to 1
        while (p <= 1.01) {
                // reliability of each link
                double reliabilities[numEdges];
                setReliabilityOfEdge(reliabilities, numEdges, p);
                // get the reliability of the system
                double reliability = calculateReliability(numNodes, numEd
                printf("%lf \t %lf\n", p, reliability);
                p = p + 0.05;
        }
```

```
// case 2: fix p to 0.9, run 5 times for each k in [0, 20]
// get all possible combinations of given nodes and edges
// pick any random k combinations and flip the states
// calculate reliability
p = 0.9;
llist_st *list = NULL;
int combinations = 0;
double reliabilities[numEdges];
double reliability = 0;
setReliabilityOfEdge(reliabilities, numEdges, p);
for (i = 0; i <= numEdges; i++) {
        llist_st *temp = getCombinations(0, numEdges-1, i);
        llist_st *temp1 = temp;
        while (temp1->next != NULL) {
                temp1 = temp1->next;
                combinations ++;
        }
        combinations++;
        temp1->next = list;
        list = temp;
}
list = reverseList(list);
llist_st *temp = list;
while (temp != NULL) {
        if (isSystemDown(numNodes, temp->c, numEdges, edges) == 0)
                reliability = reliability + getProbability(temp->c
        }
        temp = temp->next;
}
printf("Total combinations: %d Reliability: %lf\n", combinations,
int k = 0;
for (k = 0; k <= 20; k++) {
        int j = 0;
        double total_reliability = 0;
        for (i = 0; i < 1000; i++) {
                double local_reliability = reliability;
                for (j = 0; j < k; j++) {
                        int seq = rand()%combinations;
                        llist_st *comb = getSequence(list, seq);
```

```
                                char *comb_c = comb->c;
                                double prob = getProbability(comb_c, numEd
                                if (isSystemDown(numNodes, comb_c, numEdge
                                        local_reliability += prob;
                                } else {
                                        local_reliability -= prob;
                                }
                                local_reliability = local_reliability<0?0:
                        }
                        total_reliability = total_reliability + local_relia
                        total_reliability = total_reliability<0?0:total_re
                }
                //printf("k: %d Reliability: %lf\n", k, total_reliability/
                printf("%lf\n", total_reliability/1000);
        }
        free_list(list);
}
```

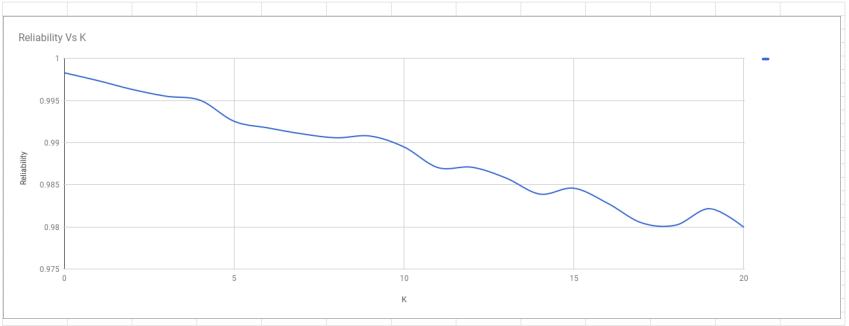

Figure 1: K vs Reliability

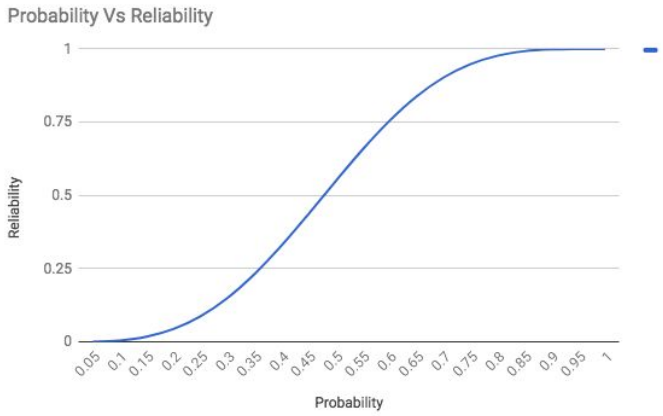

Figure 2: Probability vs Reliability