Implementation of Polynomial Time Algorithms for Network Coding – with Python language

Orlovic Dusan, PhD student, University of Novi Sad

***Abstract* — This work shows one implementation of polynomial time algorithms in centralized linear network coding. Used programming language is Python, language which can show simplicity and efficiency of such algorithms. Program was tested with random graphs and some user defined graphs.**

***Index terms* — network coding**

I. Introduction

NETWORK CODING is concept about code construction for network flows, that is next step from classical «receive and send» routing. It allows intermediate nodes in network to combine information they receive, i.e. to merge input data into one symbol which will fulfill requirements of node neighborhoods **within one step**. In past several years this idea became developed in strong theory [1] [2].

At beginning, it is important to show tology where Network coding can be profitable. Baochun Li [3] has made interesting video presentation about «How helpful is Network Coding» where he had shown several limitations on which must be worked in future (synchrony, delay, CPU usage). He explained that Coding Advantage CA (the ratio of the best throughput with network coding over that without coding) is only one in unicast and broadcast sessions in directed networks (no coding advantage). In multicast sessions CA is upper bounded with two, hence, network coding is useful only in multicast sessions and only two times better than without coding.

Network coding brings good news in problems of maximizing the capacity in multicast sessions, which is, without coding, at least hard as the minimum directed Steiner tree problem [4], thus it is NP-hard to even approximate the maximum rate. With network coding, maximum rate is ease to calculate as minimum of mincuts to all sinks and following algorithm gives solutions in polynomial time.

II. Polynomial algorithms

A. Related Work

Ahlswede et al. [5] introduce a term Network coding as necessary tool for Butterfly problem (where edges contain more than one flow). They say that, as field size approaches infinity it is possible to maximizate throughput to h (h is min of mincuts to all sinks). Li et al. [6] show that field size can be finite for linear coding. Rasala-Lehman and Lehman [7] give lower bounds on the minimum alphabet size and proved that finding the smallest size is NP-hard. Koetter and Medard [8] show that $|F|=\Theta(|T|\cdot h)$ ($|T|$ is number of sinks) is enough for the field size, and Ho et al. [9] give the similar result using randomized approach. They give the probability that random linear network code achieves mincut rate and it tends to one as ratio of number of sinks and field size $|T|/|F|$ tends to zero. They also noted that it could be realized in distributed scenario.

On the other side Jaggi et al. [10] give polynomial time algorithms with field size $\Theta(|T|)$ for **centralized** network code construction which are used in this work.

B. Algorithm description

Consider unit capacity, multiedges directed graph *G(V,E)*. (nonunit capacity edges are replaced with multi unit capacity edges). This centralized algorithm has two parts. First, flows are determined for each sink $t \in T$ and edges without flows are discarded (notice that flows for the same sink doesn't have mutual edges). Mincut h is evaluated as min of number of flows for each sink. All flows above first h flows are also discarded. Second part of algorithm is to calculate edge vectors, which are used to represent data over it.

Example: information data $\vec{x}$ is h dimensional vector over some field $\vec{x} = [x_1, x_2, \ldots x_h] \quad x_i \in F$. Field size is 2^m so data can be viewed as bits. Edge data y on edge e is symbol from F obtained from scalar product of information data and h dimensional vector (also called global vector) on edge e: $\vec{b_c} = [b_{c,1}, b_{c,2}, \ldots b_{c,h}]$

$$y = \vec{x} \cdot \vec{b_c} = x_1 b_{c,1} + x_2 b_{c,2} + \ldots + x_h b_{c,h} \qquad (1)$$

Vector on edge e originating at some vertex is linear combination of input vectors (that are vectors on edges ending at that vertex and having mutual flows with e).

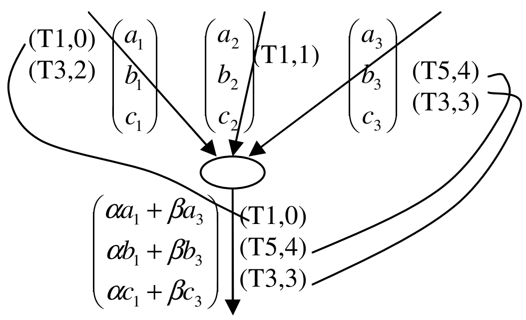

Fig. 1 Calculating edge vector *e*

Dušan Orlović, Faculty of Technical Science at University of Novi Sad. Advisors: dr Vukobratovic Dejan, dr Senk Vojin. Fellowship holder of Ministry of Science in Republic Serbia (email: orlovic@uns.ns.ac.yu)

On Fig. 1 (Ti , j) is label of j-th flow to Ti sink. In this example, output edge contains flows to (T1,0), (T5,4) and (T3,3) therefore linear combination take into consideration only first and third input edge. (T1,1) flow on second edge doesn't go through edge e.

Size of linear combination $[\alpha,\beta,...]$ is number of input edges having common flows. Linear combination must ensure that sinks can resolve h symbols from their inputs (as each flow carry one dimension from the total h dimensions, linear calculation must not suspend any flow).

In other words, if we put edge vectors of *h* flows to certain sink *t* at columns in (h x h) matrix B_t, algorithm invariant is that B_t is always invertible and therefore information data could be retrieved. In (2) $\vec{x}$ is information vector and $\vec{y}$ is vector of symbols on c_1, c_2, ...c_h edges (current edges on flows).

$$\vec{x}\cdot B_t = [x_1,..x_h]\cdot\begin{bmatrix} b_{c_1,1} & b_{c_i,1} & b_{c_h,1} \\ b_{c_1,i} & & \\ b_{c_1,h} & & b_{c_h,h} \end{bmatrix} = [y_1,..y_h] = \vec{y}_t \qquad (2)$$

At sinks information vector can be retrieved with formula $\vec{x} = \vec{y}_t\cdot B_t^{-1} = \vec{y}_t\cdot A_t$ where A_t is inverse matrix. Note its rows as vectors $\vec{a}_i$. Vector $\vec{a}_i$ is perpendicular on space B/{b_i} because of (3) (we will use this extensively for testing if new vector is appropriate):

$$A\cdot B = I \;\Rightarrow\; \vec{a}_i\cdot\vec{b}_j = \begin{cases} 0 & if\ i\neq j \\ 1 & if\ i = j \end{cases} \qquad (3)$$

In algorithm, usual matrix inversion is avoided using Sherman-Morrison formula [11]. It helps because in each step we are changing only the one column in B_t (not the whole matrix) for example i-th column becomes $b=[b_1,b_2,...b_h]$. It is the same as we add to B some matrix λ which has only i-th column different from zero and equal $-b_{ci} + b = [-b_{ci,1}+b_1,\ -b_{ci,2}+b_2\ ...\ -b_{ci,h}+b_h]$. Main formula for updating inverse matrix is derived from the following equation (A' prim means new inversion matrix).

$$A' = (B+\lambda)^{-1} = \left(B\left(1+B^{-1}\lambda\right)\right)^{-1} = (1-A\lambda+A\lambda A\lambda-...)A$$

$$=(1-A\lambda(1-A\lambda+A\lambda A\lambda-...))A = \left(1-\frac{A\lambda}{1+A\lambda}\right)A$$

Those components are simple to calculate.

$$A\lambda = \begin{bmatrix} a_{1,1} & ... & a_{1,h} \\ a_{i,1} & ... & a_{i,h} \\ a_{h,1} & ... & a_{h,h} \end{bmatrix}\begin{bmatrix} 0 & -b_{c_i,1}+b_1 & 0 \\ 0 & -b_{c_i,i}+b_i & 0 \\ 0 & -b_{c_i,h}+b_h & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \vec{a}_1\left(-\vec{b}_{ci}+\vec{b}\right) & 0 \\ 0 & \vec{a}_i\left(-\vec{b}_{ci}+\vec{b}\right) & 0 \\ 0 & \vec{a}_h\left(-\vec{b}_{ci}+\vec{b}\right) & 0 \end{bmatrix} \overset{(3)}{=} \begin{bmatrix} 0 & \vec{a}_1\vec{b} & 0 \\ 0 & \vec{a}_i\vec{b}-1 & 0 \\ 0 & \vec{a}_h\vec{b} & 0 \end{bmatrix}$$

$$(1+A\lambda)^{-1} = \begin{bmatrix} 1 & \vec{a}_1\vec{b} & 0 \\ 0 & \vec{a}_i\vec{b} & 0 \\ 0 & \vec{a}_h\vec{b} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/\vec{a}_i\cdot\vec{b} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\det(1+A\lambda) = \vec{a}_i\cdot\vec{b}$$

Finally we have:

$$A' = \left(1-A\lambda\cdot(1+A\lambda)^{-1}\right)A = \begin{bmatrix} 1 & -\vec{a}_1\vec{b}/\vec{a}_i\vec{b} & 0 & 0 \\ 0 & 1/\vec{a}_i\vec{b} & ... & ... \\ ... & ... & 1 & 0 \\ 0 & -\vec{a}_h\vec{b}/\vec{a}_i\vec{b} & ... & 1 \end{bmatrix}A$$

$$= \begin{bmatrix} a_{1,1}-\frac{\vec{a}_1\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,1} & a_{1,2}-\frac{\vec{a}_1\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,2} & ... & a_{1,h}-\frac{\vec{a}_1\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,h} \\ a_{i,1}/\vec{a}_i\vec{b} & a_{i,2}/\vec{a}_i\vec{b} & ... & a_{i,h}/\vec{a}_i\vec{b} \\ ... & ... & & ... \\ a_{h,1}-\frac{\vec{a}_h\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,1} & a_{h,2}-\frac{\vec{a}_h\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,2} & ... & a_{h,h}-\frac{\vec{a}_h\vec{b}}{\vec{a}_i\vec{b}}\cdot a_{i,h} \end{bmatrix}$$

Update formula for the inverse matrix A can be derived from the last equation (recall that we were updating i-th column of the matrix B with vector *b*):

$$\begin{aligned} \vec{a}_i' &= \vec{a}_i/\left(\vec{a}_i\cdot\vec{b}\right) \\ \vec{a}_j' &= \vec{a}_j-\vec{a}_i'\left(a_j\cdot b\right) \end{aligned} \qquad (4)$$

The main step in algorithm is to chose appropriate linear combination (see Fig. 1) which will assure h dimensionality of matrix B. To test if a new vector *b* on i-th position is a good one, we are performing only one calculation: multiplying new vector *b* with a perpendicular on B/{bi} (i.e. vector a_i) and if we got a number different than zero, we had chosen right b.

$$\vec{b}\cdot\vec{a}_i\begin{cases} =0 & new\,b\ depends\ on\ B/\{b_i\},\ so\ try\ again \\ \neq 0 & new\,b\ has\ projection\ on\ perpendicular \\ & hence\ h\ \mathrm{dim}ensionality\ is\ preserved \end{cases}$$

Randomized algorithm, where coefficients in linear combination are random numbers from finite field, has expected running time $O = \left(|E||T|h^2\right)$. Most of the operations are spent for updating inverse vectors and they occupy multiplication of: |E| (iterate over all edges), |T| (at the most |T| flows), *h* (h is number of inverse vectors for every flow) and *h* (vector dimension = inner product complexity). Failure probability is |F|/|T| (here is used field size of $|F|\geq 2|T|$). Deterministic algorithm is based on this reason: if there is n≤|F| pairs of non normal vectors

(b_i, a_i) (i.e. $b_i a_i \neq 0$) then there exists linear combination u of $b_1 \ldots b_n$ such that $ua_i \neq 0$ (linear combination isn't normal to any a_i). Algorithm steps are: $u_1 = b_1$, $1 \leq i < n$ if $u_i a_{i+1} \neq 0$ then $u_{i+1} = u_i$ else $u_{i+1} = \alpha u_i + b_{i+1}$ where α is from finite field except set$\{\alpha_k \mid \alpha_k = -(b_{i+1} a_k)/(u_i a_k)\ 1 \leq k \leq i\}$, it has running time $O(|T|^2 h)$ (at most $|T|$ pairs, h for calculating α and size of admissible α is at most $|T|$) Deterministic running time is $O(|E|(|T|h^2 + |T|^2 h)) = O(|E||T|h(h + |T|))$ and required field size is any $|F| \geq |T|$ For the source code see [12]. Each sink can reconstruct all h input symbols in $O(h^2)$.

C. Python implementation

Algorithm realization is done with Python interpreter language, which is the most suitable language for such testing. Whole implementation could be written on one page as it is in this work in APPENDIX. That Python code with detailed installation instructions and examples can be found at [12]. Program imports several modules: NetworkX for graph operations, Numpy for vector operations, Pydot for drawing graphs and ffield for finite field operations.

Program input is directed acyclic, multigraph *(V,E).* Multigraph can be defined in txt file dot format (edges and capacity) or chosen randomly. If there is non unit integer capacity between nodes example: c=3, it is replaced with c multi edges with unit capacity.

Source node is node without any input edges (if graph doesn't contain only one such node, program exits) and sinks are nodes without output edges.

Next step is to mark all flows to all sinks (example: for sink T0, there are three flows (T0,0), (T0,1), (T0,2)). Mincut *h* is min of number of flows to each sink. Remaining flows above first *h* are discarded and edges without flows are also discarded. Then, program inserts new *h* pseudo edges from pseudo node 's' to the source, with vectors (1,0,0,0,...), (0,1,0,0,0,...) ... (0,0,....0,1) (*h* vectors) respectively, and mark them as *current edges* for *h* flow, for all sinks. Initially *perpendiculars* are the same as *vectors* because it holds (see equation (3)).

$$A_t \cdot B_t = \begin{array}{cc} & \\ & \begin{bmatrix} 1 & 0 & & 0 \\ 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{hxh} \end{array} \begin{array}{c} \\ a_i \\ \\ \\ \end{array} \begin{array}{cccc} & b_i & & \\ \end{array} \begin{bmatrix} 1 & 0 & & 0 \\ 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{hxh} = I$$

Nodes are processed in topological order (it means that all input edges were processed before node is processing). When an edge e is processing, first are determined all vectors and perpendiculars on input edges having the same flows as e has (see Fig. 1). Note those vectors as *temp_vectors* and *temp_perpendiculars*, respectively. Label *lin_comb* is linear combination of vectors from *temp_vectors* with coefficients from finite field. *field* size is of form 2^m and at least double of number of sinks. Scalar product (label *multiplication*) of vectors in *temp_perpendiculars* and *lin_comb* must be non zero. This ensure that new vector *lin_comb* have component at perpendicular on B_t without vector which *lin_comb* is replacing. Rest of algorithm is updating *current_edges, vectors* and *perpendiculars.* At the end, picture of the graph with edge flows and vectors, is created.

D. Testing

This program gives us ability to find centralized network code solution to any multicast session. Fig. 2 presents a solution of butterfly problem with double edges (mincut is therefore 4). In Fig. 3 is presented deterministic solution for random generated network (see [12] for details of generation steps). (*h* is 5, sinks are {8, 9}, field size is 2)

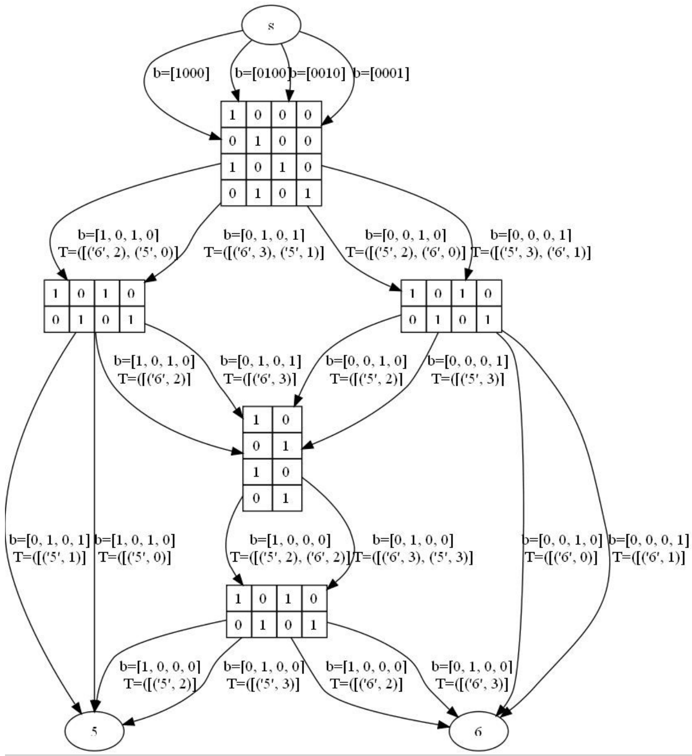

Fig. 2 Butterfly network with double capacity edges

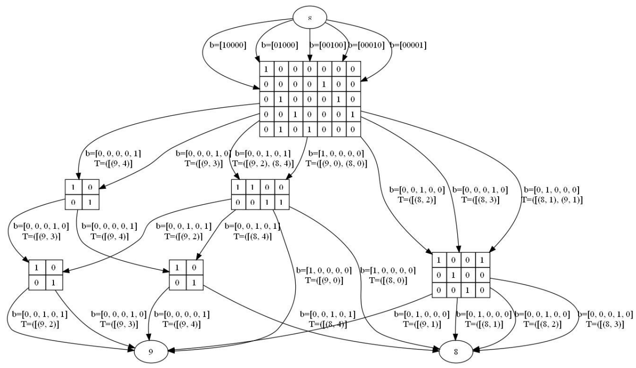

Fig. 3 Random network with 10 nodes

III. Conclusion

This work shows simplicity of modern algorithms in network coding [10] and attracts young scholars to involve in this new exciting mixture field of information coding, graphs theory, routing etc.

Appendix

Source code for randomized algorithm:

```
import ffield #www.mit.edu/~emin/source_code/py_ecc/ffield.py
import numpy #download from http://numpy.scipy.org/
import networkx # http://pypi.python.org/pypi/networkx/
import math
import random
import pydot #http://code.google.com/p/pydot/

picture = pydot.Dot()
graph = networkx.read_dot('dot.txt')
graph.ban_selfloops()
for h,t,c in graph.edges()[:] :
    if c!={} and c != None:
        capacity = int(c.pop('c'))
        if capacity > 1:
            for x in range(capacity-1):
                graph.add_edge(h,t,{})
        elif capacity ==0:
            graph.delete_edge(h,t,c)
if graph.in_degree().count(0) > 1:
    exit("ERROR: The Graph has more than one vertice with no input edges (the problem is more than one sources)!")
elif graph.in_degree().count(0) < 1:
    exit("ERROR: The Graph doesn't have any vertice with no input edges (the problem is that there is no source)!")
source=graph.nodes()[graph.in_degree().index(0)]
sinks=[graph.nodes()[i] for i,x in enumerate(graph.out_degree()) if x==0]

new_graph = networkx.XDiGraph(multiedges=True)
flows=dict(zip(sinks,[list() for dump in range(len(sinks))] ))
for sink in sinks:
    temp_graph=graph.copy()
    flow = networkx.path.shortest_path(temp_graph,source,sink)
    flow_number=-1
    while flow:
        flows[sink].append(flow)
        flow_number +=1
        temp_graph.delete_edges_from(zip(flow[:-1],flow[1:],[{}]*(len(flow)-1)))
        for h,t in zip(flow[:-1],flow[1:]):
            if new_graph.has_edge(h,t):
                for temp in new_graph.get_edge(h,t):
                    if set([x for x,dump in temp]).intersection([sink]) == set([]):
                        temp.add((sink,flow_number))
                        break
                else:
                    new_graph.add_edge(h,t,set([(sink,flow_number)]))
            else:
                new_graph.add_edge(h,t,set([(sink,flow_number)]))

        flow = networkx.path.shortest_path(temp_graph,source,sink)

mincut=min([len(x) for x in flows.values()])

for sink in sinks:
    while len(flows[sink]) > mincut:
        flow_number= len(flows[sink]) -1
        flow = flows[sink].pop()
        for h,t in zip(flow[:-1],flow[1:]):
            for temp in new_graph.get_edge(h,t):
                if temp.intersection(set([(sink,flow_number)])) != set([]):
                    temp.discard((sink,flow_number))
                    if len(temp) == 0:
                        new_graph.delete_edge(h,t,set())
                    break
graph=new_graph

field=ffield.FField(int(math.ceil(math.log(len(sinks),2))))
vectors=dict(zip([x for x in sinks],[[numpy.array([ffield.FElement(field,0)]*i+[ffield.FElement(field,1)]+[ffield.FElement(field,0)]*(mincut-i-1)) for i in range(mincut)] for dump in range(len(sinks))]))

for i in range(mincut):
    graph.add_edge('s',source,set([(t,i) for t in sinks]))
    picture.add_edge(pydot.Edge('s',str(source),label="b=["+'0'*i+'1'+'0'*(mincut-i-1)+"]"))

current_edges = dict(zip(sinks,[[('s',source)]*mincut for dump in range(len(sinks))]))
perpendiculars=dict(zip([x for x in sinks], [[numpy.array([ffield.FElement(field,0)]*i+[ffield.FElement(field,1)]+[ffield.FElement(field,0)]*(mincut-i-1)) for i in range(mincut)] for dump in range(len(sinks))]))
for vertice in networkx.topological_sort(graph)[1:-len(sinks)]:
    label_vertice=""
    for edge in graph.out_edges_iter(vertice):
        temp_perpendiculars = []
        index_temp_perpendiculars = []
        temp_vectors = []
        temp_orders = []
        for edge_order,input_edge in enumerate(graph.in_edges(vertice)):
            same_flows = set.intersection(input_edge[2],edge[2])
            if same_flows:
                for t,i in same_flows :
                    temp_perpendiculars.append(perpendiculars[t][i])
                    index_temp_perpendiculars.append((t,i))
                temp_vectors.append(vectors[t][i])
                temp_orders.append(edge_order)

        got_linear_combination = False
        while not got_linear_combination:
            lin_comb = numpy.array([ffield.FElement(field,0)]*mincut)
            random_vector = [ffield.FElement(field,random.randint(0,pow(2,field.n)-1)) for dump in range(len(temp_orders))]
            for v in [x*y for x,y in zip(temp_vectors, random_vector)]:
                lin_comb += v

            multiplication = temp_perpendiculars * lin_comb
            for i in multiplication:
                if i.sum().f == 0:
                    print "random_vector", random_vector,"lin_comb", lin_comb," have sum=0"
                    break
            else:
                print  "global vector on edge",edge[0],'-',edge[1],"is",lin_comb
                got_linear_combination = True
                for t,i in edge[2]:
                    vectors[t][i]=lin_comb
                    current_edges[t][i] = (vertice,edge[1])
                    index=index_temp_perpendiculars.index((t,i))
                    perpendiculars[t][i] = temp_perpendiculars[index] * ffield.FElement(field,field.Inverse((temp_perpendiculars[index]*lin_comb).sum().f))
                    for x,temp in enumerate(perpendiculars[t][:]):
                        if x==i:
                            continue
                        temp += perpendiculars[t][i] * (lin_comb*temp).sum()
                    label=""
                lin_comb_coef=zip(temp_orders, random_vector)
                lin_comb_coef.append(('x',0))
                lin_comb_coef.reverse()
                position,coeficient = lin_comb_coef.pop()
                for i in range(graph.in_degree(vertice)):
                    if position==i:
                        label+=str(coeficient.f)+"|"
                        position,coeficient = lin_comb_coef.pop()
                    else:
                        label+="0|"
                label="{"+label[:-1]+"}"
                label_vertice +=label + "|"
                picture.add_edge(pydot.Edge(src=str(vertice),dst=str(edge[1]),label="b="+str([x.f for x in lin_comb]) + "\\nT="+ str(edge[2])[3:-1] ))
    if label_vertice != '':
        picture.add_node(pydot.Node(str(vertice),label=label_vertice[:-1],shape='record'))

picture.write_jpg('dot.jpg')
```

References

[1] Yeung, Li, Cai, Zhang “Network Coding Theory” Foundations and Trends® in Communications and Information Theory: now Publishers Inc, 2005.

[2] Tracey Ho, Desmond S. Lun “Network Coding: An introduction” Cambridge University Press, 2008.

[3] Baochun Lin, Microsoft research speaker at ResearchChannel.org http://www.researchchannel.org/prog/displayevent.aspx?rID=6940&fID=345 2006.

[4] K. Jain, M. Mahdian and M.R. Salavatipour “Packing Strainer Trees” in Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA) Baltimore, MD, Jan 2003.

[5] Ahlswede, Cai, Li, Yeung,”Network information flow” *IEEE Trans. Inf. Theory, vol. 46, no. 4, pp. 1204-1216, Jul 2000.*

[6] Li, Yeung, Chai, “Linear network coding”, *IEEE Trans. Inf. Theory, vol. 49, no. 2, pp. 371-381, Feb. 2003.*

[7] A. Rasala-Lehman and E. Lehman “Complexity classification of network information flow problems”, SODA ’04: Proc: 15th Annu. ACM-SIAM Symp. Discrete algorithms, New Orleans, LA 2004 pp. 142-150.

[8] R. Koetter, M. Medard, “An algebraic approach to network coding”, IEEE/ACM Trans. Netw. Vol.11, no. 5, pp. 782-795, Oct. 2003.

[9] T. Ho, R.Koetter, M. Medard, D. Karger, M. Effros “The benefits of coding over routing in a randomized setting” in Proc. IEEE Int. Symp. Inf. Theory ISIT, Yokohama, Japan, Jun 2003, p.442

[10] S. Jaggi, P. Sanders, P.A. Chou, M. Effros, S. Egner, K. Jain, L.M.G.M Tolhuizen “Polinomial time algorithms for multicast network code construction” IEEE Trans. Inform. Theory, vol. 51, no. 6, pp. 1973-1982, Juna 2005.

[11] W.H. Press, S.A. Teukolsky, W. T. Vetterling, B.P. Flannery “Numerical Recipes in C, 2nd” Cambridge, U.K. 1992, section.2.7.

[12] http://sites.google.com/site/duleorlovic/solutions/implementation-of-polynomial-time-algorithms-for-network-coding.