IEEE Information Theory Workshop, Bergen, Norway, 2007.

A Decentralized Approach to Network Coding Based on Learning

Mohammad Jabbarihagh, Farshad Lahouti
Wireless Multimedia Communications Laboratory
School of ECE, University of Tehran
m.jabari@ece.ut.ac.ir, lahouti@ut.ac.ir

***Abstract*—Network coding is used to efficiently transmit information in a network from source nodes to sink nodes through intermediate nodes. It has been shown that linear coding is sufficient to achieve the multicast network capacity. In this paper, we introduce a method to design capacity achieving network codes based on reinforcement learning and makes using the market theory concepts. We demonstrate that the proposed algorithm is decentralized and polynomial time complex; while it constructs the codes much faster than other random methods with the same complexity order, especially in large networks with small field sizes. Furthermore, the proposed algorithm is robust to link failures and is used to reduce the number of encoding nodes in the network.**[1]

I. Introduction

Consider an acyclic directed graph $G = (N, E)$ in which N is the set of nodes that include the source nodes, the intermediate nodes and the sink nodes. E indicates the edges of the network, which are directed, error-free and can transmit one symbol in each transmission. The task is to multi-cast a common information from the source nodes to the sink nodes through intermediate nodes. The sources can be independent or linearly correlated.

The transmitting symbols are the elements of a finite element field $\mathbb{F}$ which is selected with respect to the number of network sinks [4]. The network capacity h is given by the Max-flow Min-cut Theorem [1] and indicates the maximum number of simultaneously transmittable symbols from the source nodes to the sink nodes.

Ahlswede *et al.* [1] show that the network capacity given by the Max-flow Min-Cut theorem is achievable with network coding. Li *et al.* [2] show that linear network coding can be used to multicast symbols at a rate equal to network capacity. Koetter and Medard [3] introduce an algebraic framework for linear coding and present a polynomial time algorithm to verify a constructed network code. Ho *et al.* [4] used this framework to show that linear network codes can be efficiently constructed by employing a randomized algorithm. Jaggi *et al.* [5] proposed a centralized polynomial time algorithm for constructing network codes based on deterministic algorithms and a random search. Lehman and Lehman [7] presented bounds on the coding field size. Fragouli *et al.* [8] derive code design algorithms for networks based on the graph coloring techniques.

[1]This work is supported by Iran Telecommunications Research Center (ITRC).

In this paper, we introduce a new efficient method based on reinforcement learning to construct linear network codes, referred to as the Reinforcement Learning Network Code (RLNC) design. We will make use of the market theory concepts in the learning section of the algorithm. The learning approach results in a smart random search and we demonstrate that the proposed algorithm is decentralized, and polynomial time complex. The RLNC algorithm constructs network codes faster than the random method of [4] with the same complexity order, especially in large networks with small field sizes. It is also shown that the algorithm can re-construct the network code in the presence of link failures with a small complexity and without previously being aware of error patterns even in very large networks. Further extension to achieve the objective of decreasing the number of encoding nodes in a network is discussed.

II. Reinforcement Learning for Network Coding

A. Overview

Reinforcement Learning (RL) [6] is a reward-punishment based training method. In RL, the system tries different possible actions and for each of them receives a reward or punishment. Considering the history of rewards and punishments of each action, after sufficient number of trials the system learns the appropriate actions. As a result it can set a policy to choose actions in order to get the maximum reward.

The RL method is modeled based on a Markov Decision Process (MDP). Each state has actions which transfers the system to other states. The goal of RL is to find proper actions in each state to get maximum reward. RL is used in different problems, such as robot training, maze problems [6] and routing in wireless networks [9].

B. Q-learning

Q-Learning [6][10], as a Monte Carlo technique, is one of the oldest approaches to Reinforcement Learning.

Q-Learning deals with states, rewards, punishments, policy and a Q-table. The current state s_t is the system state at time t. Actions relate states to each other. Choosing an action at s_t will direct the system to the next state s_{t+1}. The rewards and punishments are feedbacks which the algorithm uses to improve the system performance. The policy π is the action selection plan and dictates which action should be chosen in each state. The Q-table stores the information about

the goodness of the actions for all states. This is done by maintaining the Q-values $Q(s,a)$ for each state-action pair, which approximates the total reward received if in the state s system chooses the action a. $Q(s,a)$ is updated every time the action a is chosen in the state s. The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a) - Q(s,a)], \quad (1)$$

in which, s and a are the current state and the action, s' is the next state, α is the learning rate, γ is the discount factor and r is reward or punishment. The value of γ $(0<\gamma<1)$ determines if the system is in an exploration or exploitation mode. The Q-table is initialized randomly at the beginning of the algorithm. Each trial is named an episode. Each episode consists of a finite number of steps. One step begins by selection of an action, observation of the corresponding reward, updating the Q-table and finally moving to the next state. If the algorithm is run for a sufficient number of episodes, the system is trained and an ultimate Q-table is produced.

C. Problem Formulation

The source node generates h symbols $(\lambda_1, ..., \lambda_h)$ and a linear combination of these symbols will be carried by different edges. We call these combinations, edge codes $c(e)$, where $c(e) = (c_1, ..., c_h), c_j \in \mathbb{F}_{2^m}, 1 \leq j \leq h$. The code of the edge emanating from the source node and carrying λ_j, is $(0^{j-1}, 1, 0^{h-j})$.

Each node n has a set of incoming edges $In(n)$ and a set of outgoing edges $Out(n)$. Each edge has end node $end(e)$, start node $start(e)$. Each code $c(e)$ is a linear combination of codes of the set of edges $In(start(e))$.

We define h data channels $(ch_1, ..., ch_h)$ for edge e, where $ch_j(e)$ corresponds to the symbol λ_j. Each channel can be *on* or *off*, when a channel is *on* it means that the edge transmits the corresponding symbol.

To solve the network coding problem using RL, we consider the set of edges as the states and channel states (*on* or *off*) as the actions. So, the Q-values are defined by $Q(edge, channel, state)$ or briefly $Q(e, ch, state)$. The algorithm examines channel state Q-values to construct the ideal codes. An ideal code $ic(e)$ with the binary vector representation $(ic_1(e), ..., ic_h(e))$, is the best possible code pattern for edge e according to the current Q-values.

In the next section ideal code construction considering Q-values and network code construction will be discussed.

III. Proposed Algorithm

A pseudo code of the algorithm is presented in Fig.1. The algorithm constructs codes for the outgoing edges of each node in a topological order. At each intermediate node n, the edge code $c(e)$ is calculated for all the edges of $Out(n)$ and the Q-values of all edges of $In(n)$ is updated. At the sink nodes, considering the decodablity of symbols different rewards are given to the incoming edges.

```
h is calculated from Max-flow Min-cut theorem
Initialize α, γ, rewards
Initialize Q-table randomly
for each e_j that start(e) = source
    b(e_j):=(0^{j-1}, 1, 0^{h-j})

Start Episode:
    Calculate ϵ
    for each node n in topological order:
      If node is intermediate:
        for each edge e' of Out(n):
            Check state of each channel from Q-values
            Find ideal code from Q-values
            Construct edge code considering ideal code
            for each edge e of In(n) and each channel ch:
                Update Q-value
      If node is sink:
        Check all incoming codes
        for each incoming edge:
            Calculate reward
            Update Q-Value
    If all sinks can decode all symbols
      end algorithm
    else
      next episode
```

Fig. 1. Pseudo code of RLNC

A. Epsilon

The parameter ϵ in Fig.1 is an indicator of the randomness of the Q-Learning algorithm and is used in the channel state selection. With probability $(1-\epsilon)\%$ the state with a higher Q-value will be selected. The parameter ϵ decreases from its initial value ϵ_init exponentially in every episode as folllows:

$$\epsilon = (\epsilon\_init)e^{\frac{episode}{total\ number\ of\ episodes}} \quad (2)$$

B. Code Construction

In code construction, at node n for each edge e, considering the Q-values of the corresponding channels $(ch_1, ..., ch_h)$, we find the ideal code $ic(e)$ as follows: If $Q(e, ch_j, on)$ is larger than $Q(e, ch_j, off)$, the bit $ic_j(e)$ with probability $(1-\epsilon)\%$ will be set to 1 otherwise will be set to 0. Then we examine all codes of edges within $In(n)$ and from there we choose suitable incoming edge codes for constructing $c(e)$ according to the following procedure. We define a code metric $d(ic(e), c(e'))$ for each $e' \in In(start(e))$ as follows:

$$d_j(ic(e), c(e')) = \begin{cases} \frac{1}{|c(e')|} & ic_j(e) = 1, c_j(e') \neq 0 \\ \frac{B}{|c(e')|} & ic_j(e) = 0, c_j(e') \neq 0 \\ 0 & ic_j(e) = 1, c_j(e') = 0 \end{cases} \quad (3)$$
$$\forall e' \in In(start(e)), j = 1, ..., h$$

$$d(ic(e), c(e')) = \sum_{j=1}^{h} d_j(ic(e), c(e')), \quad (4)$$

in which, $|c(e')|$ is the Hamming weight (one for each non zero symbol) of $c(e')$ and $-1 < B < 1$. Now, we compare the code metric $d(ic(e), c(e'))$ with a threshold T and define:

$$A = \{e' | d(ic(e), c(e')) > T, e' \in In(start(e))\} \quad (5)$$

The vector m_e is the coding vector and determines the linear combination of the edges of set A to construct $c(e)$. The coefficients of m_e are chosen uniformly at random from the finite field $\mathbb{F}_{2^m}$. We have,

$$c(e) = \sum_{e' \in A} m_e(e')c(e') \tag{6}$$

The value of B in eq.3 and T in eq.5 balance a tradeoff between the complexity and the ratio of encoding nodes. This is discussed in section VII. The constants B in eq.3, T in eq.5, α and γ in eq.7-9 and $total\ number\ of\ episodes$ in eq.2 are learning parameters and are defined after several observations. In practice, we assume $B = 0.25$, $T = 0.4$, $\alpha = 0.7$, $\gamma = 0.9$ for network coding and the value of $total number of episodes$ for different networks is provided in Tabel I.

C. Q-table update

At the intermediate node n, the Q-values of each edge $e \in In(n)$ is updated considering edges $e' \in Out(n)$ one by one. Following eq.1, this is done a follows:

$$\begin{aligned} Q(e, ch_j, state) \leftarrow\ & Q(e, ch_j, state) \\ & + \alpha[\gamma \max_{state} Q(e', ch_j, state) \\ & - Q(e, ch_j, state)], \end{aligned} \tag{7}$$

where $e \in In(n)$ and $e' \in Out(n)$.

At the sink node, the Q-values of all edges of $In(sink)$ are updated considering the rewards $r(e, ch_j, state)$, as qualified in next subsection. Following eq.1, we have:

$$\begin{aligned} Q(e, ch_j, on) \leftarrow\ & Q(e, ch_j, on) \\ & + \alpha[r(e, ch_j, on) - Q(e, ch_j, on)] \end{aligned} \tag{8}$$

$$\begin{aligned} Q(e, ch_j, off) \leftarrow\ & Q(e, ch_j, off) \\ & + \alpha[r(e, ch_j, off) - Q(e, ch_j, off))], \end{aligned} \tag{9}$$

where $e \in In(sink)$.

D. Reward and Punishment

For the assignment of the reward and punishment, we use the market theory concepts of supply and demand. This is only done for the edges which terminate in the sink nodes. The effect of these rewards and punishments spreads to other edges the next time they are updated considering the Q-values of the following edges.

Every time the algorithm reaches the sink nodes it checks the incoming codes and assigns reward and punishment to the incoming edges. First, the sink checks which symbols are decodable from the incoming codes. If a symbol is decodable, a reward will be distributed among the edges whose corresponding channel is *on*. This reward is referred to as the *flow reward* is divided equally between the corresponding edges. So if many edges had transmitted a common symbol they will get a small reward and if only one edge has provided the symbol it will get a large reward. This is exactly a supply and demand situation. At the same time a reward named *enough reward* will be given to the *off* channel of other edges, because the symbol could be decoded at the sink and there is no need for other edges to transmit that specific symbol.

If the sink determines that it can not decode a specific symbol it will give a *demand punishment* to the Q-value of the *off* state of the corresponding symbol channel for all incoming edges. This is the situation where the supply is low and the demand is high. This *demand punishment* will encourage the edges to transmit that specific symbol.

For the $channel\ j$ of the edge e, we define the rewards $r(e, ch_j, on)$ and $r(e, ch_j, off)$ as follows:

$$r(e, ch_j, on) = \begin{cases} \frac{flow\ reward}{|G(\lambda_j)|} & c_j(e) \neq 0 \\ 0 & c_j(e) = 0 \end{cases} \tag{10}$$

$$r(e, ch_j, off) = \begin{cases} \textit{enough reward} & e \in H(\lambda_j) \\ \textit{demand punishment} & e \in J(\lambda_j), \\ 0 & \textit{otherwise} \end{cases} \tag{11}$$

in which:

$G(\lambda_j) = \{e | end(e) = sink,\ c_j(e) \neq 0,\ \lambda_j \text{ is decodable}\}$
$H(\lambda_j) = \{e | end(e) = sink,\ c_j(e) = 0,\ \lambda_j \text{ is decodable}\}$
$J(\lambda_j) = \{e | end(e) = sink,\ \lambda_j \text{ is not decodable}\}.$

In practice, we assume the value of *flow reward=100*, *demand punishment=–150* and *enough reward=10*, based on several experiments.

E. A Decentralized Algorithm

In RLNC, during the training phase, each node only receives/sends data from/to nodes, which are directly connected to it. Therefore, in order to find a proper code for a specific node no information is needed from all other nodes or a central control. In fact, code construction in each node is based on the incoming codes, the ideal code and the channel states. With the exception of the incoming codes, all other data mentioned above are extracted from the Q-values. In the Q-table update rules, eq.7-9, it is obvious that the Q-values are updated only considering the Q-values of the outgoing edges and rewards. This indicates that, using the proposed algorithm, the design of the network code is performed in a completely decentralized manner, as is the case for the original Q-Learning algorithm.

IV. Complexity Analysis

In Reinforcement Learning there are three kinds of complexity: space complexity, computational complexity and sample complexity [13]. The space complexity measures the amount of memory required to implement the algorithm. The computational complexity measures the amount of operations needed in each episode. The sample complexity measures the number of episodes until the algorithm behaves nearly optimal, or i.e. the amount of experience it takes to learn to behave well.

A. Space Complexity

All data of the algorithm is stored in the Q-table, so the space complexity has a direct relationship with the Q-table size. Our Q-table has tree dimensions: edges, channels and states. Therefore, the table size is $2Eh$ and the space complexity is $O(Eh)$.

B. Computational Complexity

At the intermediate nodes, for each edge, building ideal code $ic(e)$ takes time $O(h)$. The calculation of code distances is done in $O(|In(n)|h)$ and the construction of the code $c(e)$ takes time $O(|In(n)|h)$. Combining all edges, $|In(n)|$ is summed to $|E|$ and results in a complexity of $O(|E|^2h)$ for the code construction in one episode. At each sink finding all decodable symbols takes time $O(h^2)$ in worst case. The assignment of rewards and the Q-table update also take time $O(h^2)$. Therefore, the sink-check part of the episode is done in $O(|T|h^2)$. Overall, each episode runs in time $O(|E|^2h+|T|h^2)$ (computational complexity). This is the same order of computational complexity of each trial or run in [4].

C. Sample Complexity

Computing sample complexity of RL methods in general is still an open problem [13]. Some algorithms which their sample complexity has been determined to be polynomial time, suffer from high computational complexity such as R_{max}, E^3 or $MBIE$ [13]. Recently, Strehi et al. [13] introduced a new algorithm based on Q-learning, dubbed Delayed Q-Learning, which has low computational complexity and polynomial time sample complexity. In general the suggested algorithms fit specific scenarios, and may not be directly applicable to other problems such as network coding. In some cases, statistical methods are used to determine the sample complexity of the RL algorithms [9]. This is the approach we take in this work.

To analyze the sample complexity, we setup two sets of simulations.[2] The simulations study the dependency of the sample complexity with different network architecture attributes. Two main concepts in the simulations are the capacity and connectivity. the connectivity is the average number of the outgoing edges of the network nodes and is defined as direct relationship with the number of edges. It is obvious that the capacity has a direct relationship with the number of nodes.

All simulations are done in the binary field. At the end of each episode to measure the distance from the optimal codes, we use the normalized Hamming distance (NHD). For each of the edges, we calculate the hamming distance between its current code and the optimal code of that edge. Adding these distances and dividing the result to number of edges we get NHD of the current codes.

We discuss the simulations in two cases:

1) Effect of number of edges on sample complexity: In this case the number of nodes is constant and the connectivity (or the number of edges) is varied. The networks all have 109 nodes. The connectivity is changed from 2 to 9 and the edges from 230 to 895. Fig.2 shows the average NHD for 200 algorithm runs on 100 different such networks. From Fig.3 it is obvious that when the number of nodes is constant, the connectivity does not affect the sample complexity or the convergence rate.

[2]All simulation source codes can be found at: http://wmc.ut.ac.ir/Students/jabbari.htm

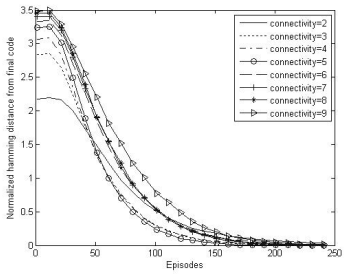

Fig. 2. Average NHD for different connectivities (N=109)

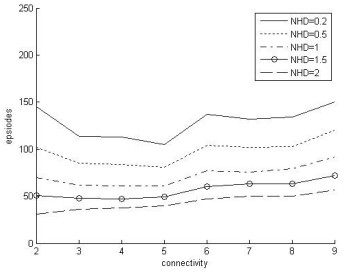

Fig. 3. Average number of episodes to converge to specific NHD targets for different connectivities (N=109)

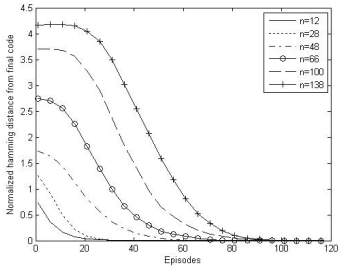

Fig. 4. Average NHD for different network sizes

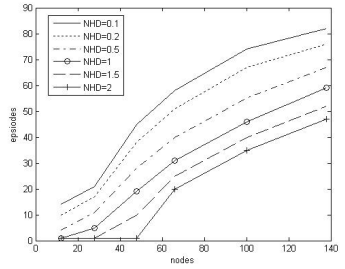

Fig. 5. Average number of episodes to converge specific performance targets for different network sizes

2) Effect of number of nodes on sample complexity: In this case, the number of nodes is varied. In order to eliminate the impact of the number of edges from the simulations we maintained a constant capacity/connectivity ratio. The capacity/connectivity ratio is 1.5 and the networks have capacities from 3 to 9 symbols and 12 to 138 nodes. Fig.4 shows average NHD of 200 algorithm runs on 100 different networks. From Fig.5 it is obvious that there is a linear relationship between the number of episodes to converge specific performance targets and the number of nodes.

V. Field Size

In [4] a minimum field size of $|T|$, which indicates the number of sinks, is considered that guarantees the existence of a capacity achieving linear network code. In general, linear network codes may be constructed on smaller fields. In [7], it is shown that there exist, networks that are not solvable on fields smaller than $\Omega(\sqrt{|T|})$.

Like all other methods of random code assignment, the RLNC constructs network codes faster, when its field size increases. We evaluated code construction run time for different network and field sizes through simulations. We also simulated the work of Ho *et al* [4], which has the same complexity order with the RLNC. Table I shows the simulation results. Although in some cases the field size is smaller than $|T|$, all the networks considered are solvable in these fields, in the sense that, a corresponding code can be found in practically large number of episodes. The simulation is done on 30 networks for 30 instances of network code construction each. As expected the RLNC run-time (number of episodes) is less than that of random coding [4]. This is because in each episode the proposed RLNC uses data and Q-values from previous episodes in order to design the code more efficiently. This is why we refer to it as a "smart random search". Specifically, for large networks and small field sizes (situations of practical interest), the proposed algorithm constructs the codes more efficiently.

TABLE I
Run time of two algorithms for different filed sizes

Network	Field size	Random Coding	RLNC
C=4	2	11.32	6.02
N=19	2^2	2.11	2.01
E=48	2^3	1.35	1.33
S=2	2^4	1.15	1.14
C=8	2	461.3	29.26
N=81	2^2	4.44	4.31
E=448	2^3	1.83	1.82
S=4	2^4	1.31	1.31
C=10	2	598.2	47.13
N=134	2^2	6.35	6.52
E=1004	2^3	2.13	2.12
S=5	2^4	1.41	1.40

C: Capacity, N: Nodes, E: Edges, S: Sinks

VI. Link Failure

The Q-table store information for all possible codes and each edge can construct best suitable code in any state like when one of its incoming edges is failed. So, unlike other approaches in network coding [5], the algorithm does not need to know the error patterns beforehand and the network can retrain itself after occurrence of any error. Since for the retraining, the Q-table data stored from the original training is utilized, the network code can be re-constructed with a lower complexity than the original code design.

We simulated failures of up to 20% of all edges in networks of different sizes, 51 to 155 node. The capacity/connectivity ratio, as described in section IV, is 1.5. For each of the network sizes 100 randomly generated networks are tested and for each of them, 100 randomly generated patterns are tested. Simulation results (not mentioned here) demonstrates that the retraining of networks in different sizes only needs 45% to 65% of the computations of the original training. Again, all simulations are done in binary field.

VII. Discussion

In order to reduce the resources used by a network code, the number of encoding nodes should be minimized. Lately, in [11] and [12] such an approach is discussed. In the proposed RLNC, to reduce the number of encoding nodes the parameters B and T in eq.3 and eq.5 are to be decreased. These parameters control the tradeoff between the convergence speed or the complexity (within the same order) and the number of encoding nodes. Decreasing parameter B will reduce the probability of involvement in a linear combination for edges that have redundant codes. Decreasing parameter T will reduce the number of edges that involve in a linear combination. Simulation results (not mentioned here) show for a set of 100 random multi-layer networks with an average of 330 edges on field $\mathbb{F}_{2^3}$, when the parameter B varies in the $(-0.75, 0.75)$ range, average percentage of non-encoding nodes decreases linearly from 65% to 15%.

VIII. Acknowledgement

The authors would like to thank Michael Littman and Majid Nili AhmadAbadi for helpful discussion on RL and complexity of RL methods.

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Trans. Inf. Theory*, vol. 46, no. 4, pp. 1204-1216, Jul. 2000.

[2] S.-Y. R. Li, R. W. Yeung, and N. Cai, "Linear network coding," *IEEE Trans. Inf. Theory*, vol. 49, no. 2, pp. 371-381, Feb. 2003.

[3] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Trans. Netw.*, vol. 11, no. 5, pp. 782-795, Oct. 2003.

[4] T. Ho, M. Médard and R. Koetter, "A random linear network coding approach to multicast," *IEEE Trans. Inf. Theory*, vol. 52, no. 10, Oct. 2006.

[5] S. Jaggi, P. Sanders, S. Egner, and P. A. Chou, "Polynomial time algorithms for multicast network code construction," *IEEE Trans. Inf. Theory*, vol. 51, no. 6, Jun. 2005.

[6] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

[7] A. R. Lehman and E. Lehman, "Complexity classification of network information flow problems," in *Proc. 15th ACM-SIAM Symp. On Discrete Algorithms*, New Orleans, LA, 2004.

[8] C. Fragouli and E. Soljanin, "Information flow decomposition for network coding," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 829-848, Mar. 2004.

[9] P. Beyens, M. Peeters, K. Steenhaut and A. Nowe, "Routing with compression in wireless sensor networks: a q-learning approach," in *Proc. Fifth European Workshop on Adaptive Agents and Multi-Agent Systems (AAMAS 05)*, Paris, 2005.

[10] C. J. C. H. Watkins, P. Dayan, "Q-Learning," *Machine Learning*, vol 8, pp. 279-292, 1992.

[11] M. Kim, C. W. Ahn, M. Médard, and M. Effros, "On Minimizing Network Coding Resources: An Evolutionary Approach," *The Second Workshop on Network Coding, Theory, and Applications (NetCod 2006)*, Boston, MA, April 2006.

[12] M. Langberg, A. Sprintson and J. Bruck, "Network Coding: A Computational Perspective," *Conf. Information Science and Systems (CISS)*, Princeton, NJ, March 2006

[13] A. L. Strehl, L. Li, E. Wiewiora, J. Langford and M. L. Littman, "PAC model-free reinforcement learning," Proc. *23rd International Conference on Machine Learning (ICML-06)*, pp 881-888, 2006.