International Journal of Uncertainty,
Fuzziness and Knowledge-Based Systems
Vol. 25, No. 3 (2017) 427–455
© World Scientific Publishing Company
DOI: 10.1142/S0218488517500180

World Scientific
www.worldscientific.com

Finding the Shortest Path in Stochastic Graphs Using Learning Automata and Adaptive Stochastic Petri Nets

S. Mehdi Vahidipour
Faculty of Electrical and Computer Engineering,
Department of Computer, University of Kashan, Kashan, Iran
School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
P.O. Box 19395-5746, Tehran, Iran
vahidipour@kashanu.ac.ir

Mohammad Reza Meybodi
Soft Computing Laboratory, Computer Engineering and Information Technology Department,
Amirkabir University of Technology (Tehran Polytechnics),
424, Hafez Ave, Tehran, Iran

Mehdi Esnaashari
Computer Engineering Department,
K. N. Toosi University of Technology, Tehran, Iran

Received 15 July 2015
Revised 23 September 2016

Shortest path problem in stochastic graphs has been recently studied in the literature and a number of algorithms has been provided to find it using varieties of learning automata models. However, all these algorithms suffer from two common drawbacks: low speed and lack of a clear termination condition. In this paper, we propose a novel learning automata-based algorithm for this problem which can speed up the process of finding the shortest path using parallelism. For this parallelism, several traverses are initiated, in parallel, from the source node towards the destination node in the graph. During each traverse, required times for traversing from the source node up to any visited node are estimated. The time estimation at each visited node is then given to the learning automaton residing in that node. Using different time estimations provided by different traverses, this learning automaton gradually learns which neighbor of the node is on the shortest path. To set a condition for the termination of the proposed algorithm, we analyze the algorithm using a recently introduced model, Adaptive Stochastic Petri Net (ASPN-LA). The results of this analysis enable us to establish a necessary condition for the termination of the algorithm. To evaluate the performance of the proposed algorithm in comparison to the existing algorithms, we apply it to find the shortest path in six different stochastic graphs. The results of this evaluation indicate that the time required for the proposed algorithm to find the shortest path in all graphs is substantially shorter than that required by similar existing algorithms.

Keywords: Stochastic graphs; the shortest path problem; learning automata; adaptive stochastic petri net based on learning automata.

1. Introduction

The deterministic shortest path problem in graphs has been studied extensively and many algorithms have been reported in the literature to solve it.1 In this problem, one looks for a path joining source and destination nodes while minimizing the summation of the traversed edges' lengths. However, there are many applications where the underlying graph is a stochastic graph and hence, the lengths of edges are random variables; a graph with stochastic edge lengths. If the probability distribution functions of these random variables are unknown, then finding the shortest path cannot be possible using the algorithms introduced for the deterministic shortest path problem. Recently, some algorithms have been proposed to find the shortest path in stochastic graphs with unknown characteristics using Distributed Learning Automata (DLA) or extended DLA (eDLA), two complex models based on the learning automaton model.[2]

Learning automaton (LA) model was introduced in 1960s3 and was popularized by Narendra.[4] An LA is an adaptive decision-making agent that improves its performance by interaction with an environment. Recently, the LA has been applied to a wide range of science and engineering applications (e.g. Refs. 1 to 4). In addition, a group of LAs can interact with each other in different interconnected structures, such as Cellular Learning Automata (CLA),[5] Irregular CLA (ICLA)6, Distributed Learning Automata (DLA),[1] and extended DLA (eDLA),[2] to solve complicated problems.

All of the DLA- or the eDLA-based algorithms introduced to solve the shortest path problem in stochastic graphs with unknown characteristics suffer from two common drawbacks. The first drawback is the low speed of these algorithms since these algorithms try to find the shortest path by sequentially traversing different paths along the graph from the source node towards the destination node, sampling the lengths of the traversed paths, and finally, identify the shortest path by comparing these sampled values. In this paper, we argue that this sequential process can be performed in a parallel manner, thus increasing the speed.

The second drawback is that in these algorithms, no clear condition is given for terminating the sequential traversing and sampling process. They use a simple condition of maximum number of traversing which indeed is not a suitable condition; the specified maximum number could be too low, resulting in inadequate number of samples, or could be too high, resulting in the wasting of time, collecting non-necessary samples. In this paper, we propose a necessary condition which if it does not hold, the number of samples collected so far will still not be enough. Thus, using AND operator between this condition and the maximum number of traversing condition can at least prevent the algorithm from being stopped before inadequate numbers of samples are collected.

In the proposed algorithm, each node i, except for the source node, is equipped with a learning automaton (LA). The LA residing in the node i is responsible to find the neighbor j of that node, which is on the shortest path from the source node to the node i. To this end, we let that several tokens, in parallel, start to traverse different paths between the source and the destination nodes, hop by hop, in such a way that each token estimates the required time for its traverse. Using the times provided by different tokens passed by, LA in each node gradually learns which neighbor is on the shortest path.

To establish a necessary condition for the termination of the proposed algorithm, it is required to first analyze the steady-state behavior of the algorithm. A suitable way of analyzing the steady-state behavior of an algorithm is to model it using a Petri net and then analyze the resulted Petri net. Therefore, in this paper, we first represent the proposed algorithm by a recently introduced stochastic Petri net, called Adaptive Stochastic Petri net based on LA (ASPN-LA7) and then analyze the steady-state behavior of the yielded ASPN-LA. The reason as to why ASPN-LA, among different kinds of Petri nets, is selected to represent the proposed algorithm is that in this model, like in the algorithm, there exists several decision points at each of which an LA is used for making decisions. Therefore, representing the algorithm by the ASPN-LA can be simply accomplished by mapping the operation of LAs in the algorithm into the operation of LAs in the decision points of ASPN-LA.

The rest of the paper is organized as follows: Section 2 gives the problem statement. Section 3 briefly reviews learning automata based algorithms introduced to solve the shortest path problem in the stochastic graphs. Section 4 gives a short review of the adaptive stochastic Petri nets. Section 5 describes the proposed algorithm. Section 6 gives the simulation results of the proposed algorithm. Section 7 concludes the paper.

2. Problem Statement

We first define a stochastic graph and then, explain the shortest path problem in stochastic graphs.

Definition 1. A stochastic graph is defined by a triple $G = (V, E, \mathcal{F})$, where $V = \{v^1, v^2, \dots, v^n\}$ is set of nodes, $E \subset V \times V$ specifies set of edges, and $n \times n$ matrix $\mathcal{F}$ is the probability distribution describing the statistics of edge lengths, where n is the number of nodes.

Here, we consider the length of an edge from node v^i to node v^j (d_{ij}) to be the time required for traversing from v^i to v^j. Since the graph is stochastic, d_{ij} is a positive random variable with f_{ij} as its probability density function; each f_{ij} is assumed to be an exponential distribution with an unknown rate parameter.

In a stochastic graph G, a path τ_i, consists of κ_i nodes and with expected length of $\bar{L}_{\tau_i}$, from a source node to a destination node is defined as an ordering $\{v_{\tau_i,1}, v_{\tau_i,2}, \dots, v_{\tau_i,\kappa_i}\} \subset V$ of nodes in such a way that $v_{\tau_i,1}$ and v_{τ_i,κ_i} are source and destination nodes, respectively, and $(v_{\tau_i,j}, v_{\tau_i,j+1}) \in E$ for $1 \leq j < k_i$, where $v_{\tau_i,j}$ is the j^{th} node in path τ_i.

Let $v^s, v^d \in V$ be the source and destination nodes, respectively. In the stochastic graph, there are $\mathcal{M}$different paths $\Omega = \{\tau_1, \tau_2, \dots, \tau_{\mathcal{M}}\}$ between v^s and v^d. Let $\Omega^l \subseteq \Omega$ be the set of loop-free paths between v^s and v^d. The shortest path between v^s and v^d denoted by $\tau^* \in \Omega^l$, is defined as a path with minimum expected length, that is $\bar{L}_{\tau^*} =$

$min_{\tau \in \Omega^l}\{\bar{L}_\tau\}$. The shortest path problem is thus to find such a path assuming no knowledge about the rate parameters of probability density functions $\mathcal{F}$.

3. Learning Automata and the Shortest Path Problem

In this section, we will briefly review LA-based algorithms introduced in the literature so far to solve the shortest path problem in stochastic graphs with unknown characteristics; the edge lengths in this graph are stochastic and come from exponential distributions with unknown rates. Before reviewing these algorithms, we briefly describe LA, DLA, and eDLA.

3.1. *Learning automata and distributed learning automata*

A learning automaton (LA) is an adaptive decision-making unit that improves its performance by learning how to choose the optimal action from a finite set of allowed actions through repeated interactions with a random environment[8]. An action is chosen at random as a sample realization of action probability distribution. The chosen action is then taken in the environment. The environment responds to the taken action in turn with a reinforcement signal. The action probability vector is then updated based on the reinforcement feedback from the environment.

An LA can be divided into two main categories[4]: fixed structure LA and variable structure LA. In what follows, variable structure LA used in this paper will be briefly described.

Definition 2. *Learning automaton* (*LA*) with variable structure is represented by $A = \{\alpha, \beta, q, L\}$, where $\alpha = \{\alpha_1, \alpha_2, \dots, \alpha_r\}$ is the set of actions, $\beta = \{\beta_1, \beta_2, \dots, \beta_m\}$ is the set of inputs, $q = \{q_1, q_2, \dots, q_r\}$ is the action probability set and L is the learning algorithm.[4]

The learning algorithm L is a recurrence relation which is used to modify the action probability vector. Let $\alpha(n)$ and $q(n)$ denote the action chosen at instant n and the action probability vector, respectively. The recurrence equations, shown by Eqs. (1) and (2), are linear learning algorithms used to update the action probability vector q. Let $\alpha(n)$ be the action chosen by the automaton at instant n

$$q_j(n+1) = \begin{cases} q_j(n) + a(n)\left[1 - q_j(n)\right], & j = i \\ \left(1 - a(n)\right)q_j(n), & \forall j \neq i \end{cases} \tag{1}$$

when the taken action is rewarded by the environment (i.e. $\beta(n) = 0$) and

$$q_j(n+1) = \begin{cases} (1 - b(n))q_j(n), & j = i \\ \left(\dfrac{b(n)}{r-1}\right) + (1 - b(n))q_j(n), & \forall j \neq i \end{cases} \tag{2}$$

when the taken action is penalized by the environment (i.e. $\beta(n) = 1$). In Eqs. (1) and (2)(2), $a(n) \geq 0$ and $b(n) \geq 0$ denote the reward and penalty parameters that determine the amount of increases and decreases of the action probabilities, respectively. If $a(n) = b(n)$, the recurrence Eqs. (1) and (2) are called linear reward–penalty (L_{R-P})

algorithm; if $a(n) \gg b(n)$, the given equations are called linear reward-ε penalty ($L_{R-\varepsilon P}$); and finally if $b(n) = 0$, they are called linear reward–Inaction (L_{R-I}). In L_{R-I}, the action probability vector remains unchanged when the taken action is penalized by the environment.

At the time instant n, an LA operates as follows: (1) it randomly selects an action $\alpha(n)$ based on the action probability vector $q(n)$, (2) it performs $\alpha(n)$ on the environment and receives the environment's response $\beta(n)$, and (3) it updates its action probability vector using the learning algorithm.

The LA is, by design, a simple unit by which simple decision makings can be performed. The full potential of the LA will be realized when a cooperative effort is made by a set of interconnected LAs to achieve the group synergy. In other words, LAs can be used as the building blocks of more complex learning models. These complex models include hierarchical system of LA,[11] distributed LA (DLA)[1], extended DLA[2], network of LA,[12] multi-level game of LA,[13,14] cellular LA (CLA)15 CLA-based evolutionary computing,[16] differential evolution-based CLA,17 CLA-based particle swarm optimization,[18] and Irregular CLA.[6] Among these complex models, DLA and eDLA has been used so far for finding the shortest path in stochastic graphs. This is because these models are better suited for sampling paths in stochastic graphs. In what follows, we will briefly describe these two models.

DLA is a network of LAs collectively cooperating to solve a particular problem. A DLA can be modelled by a directed graph where the set of nodes constitutes the set of learning automata and the set of outgoing edges for each node constitutes the set of actions for the corresponding LA. When an LA selects one of its actions, another automaton on the other end of the edge corresponding to the selected action will be activated. At any time only one LA in the network will be activated. A DLA is embedded in a graph and can be formally defined as below.

Definition 3. A *Distributed Learning Automata* (DLA) can be defined as a 4-tuple (A, E, L, A_0), where $A = \{A_1, A_2, \dots, A_n\}$ is the set of LAs, $E \subseteq A \times A$ is the set of edges in the graph where an edge (i, j) corresponds to action α_j of A_i. L is the set of learning schemes with which the LAs update their action probability vectors, and A_0 is the root LA of the DLA from which activation is started.

An extended distributed learning automata (eDLA2) is a new extension of DLA which is supervised by a set of rules governing the operation of the LAs. In general in eDLA, the ability of a DLA is improved by adding communication rules and changing the activity level of each LA. In eDLA, at any time, not only each LA can be in one mode of activity level but also each LA with a high activity level can be employed an action according to its probabilities on the random environment.

The process of finding the shortest path in DLA and eDLA, as it will be mentioned in the *Literature Review* section, is a sequential one which results in somehow slow algorithms. In this paper, we argue that a simple team of learning automata can be utilized to speed up these algorithms. The kind of learning automata which must be used in this team is a learning automaton with a variable set of actions,[9,10] where this set can be varied at any instant of time. At each time instant n, this LA creates the subset of all available actions which can be chosen, i.e. $\mathcal{A}(n)$. For all actions in $\mathcal{A}(n)$, scaled probability vector $\hat{q}(n)$ is defined as

$$\hat{q_i}(n) = q_i(n)/K(n) \tag{3}$$

where $q_i(n) = prob[\alpha(n) = \alpha_i]$ and $K(n) = \sum_{\alpha_i \epsilon \mathcal{A}(n)} q_i(n)$ is the sum of the probabilities of actions in subset $\mathcal{A}(n)$.The LA randomly selects one action from $\mathcal{A}(n)$ according to $\hat{q}(n)$. Depending on the response received from the environment, the LA updates $\hat{q}$; that is, only the probabilities of the actions in $\mathcal{A}(n)$ will be updated. Finally, the probabilities of the actions in subset $\mathcal{A}(n)$ are rescaled according to Eq. (4).

$$q_i(n+1) = \hat{q_i}(n+1) * K(n) \; if \; a_i \in \mathcal{A}(n) \tag{4}$$

3.2. *Literature review*

One available approach to solve the shortest path problem in stochastic graphs is to construct a DLA from the given stochastic graph.[12] The root LA is assigned to v^s and starts an operation of this DLA as follows:

- One of the outgoing edges of the root (one action of the root LA) is chosen using the corresponding action probability vector.
- The selected edge activates the LA at its other end. This LA also selects an action that results the activation of another LA.
- This process is repeated until the destination node v^d is reached.
- The time elapsed for this traverse from v^s to v^d is a sample of the time required for traversing from v^s to v^d.
- The sample time is compared with a quantity called ‘dynamic threshold’ which is an estimate of the required time for traversing from v^s to v^d. If the sample time is shorter than or equal to dynamic threshold, then all activated LAs get reward signal and if the sample’s time is longer than the dynamic threshold or the destination node is not reached, then all activated LAs get penalty signal. Upon the generation of reward or penalty signals, the learning algorithm L updates the action probability vectors of activated LAs.
- The value of the dynamic threshold is updated using the new sample time.

Beigy and Meybodi[1] solved the shortest path problem in a stochastic graph given a particular pair of source-destination by introducing different variations of the DLA. For example, in one variation, they assumed that the reward parameter, $a(n)$ in Eq. (1), is different in LAs assigned to different nodes of the DLA; the closer the node to the destination, the larger the value of a will be.

In Ref. 19, a version of the DLA has been defined by introducing a new definition for the dynamic threshold. The dynamic threshold is generally calculated by averaging the lengths of the paths traversed so far.[20] But in Ref. 19, the authors assumed the dynamic threshold as the minimum of the sequences of the averages, each computed at the end of one full traverse from the source to the destination node. It is argued that this new definition requires fewer number of samples to be taken from the edges of the graph to decide which path from the source to the destination node is the shortest.[21]

Mollakhalili and Meybodi[2] solved the shortest path problem in stochastic graphs using extended DLA (eDLA). To traverse a path, the activation levels of all LAs, except for the LA in v^s, are initially set to passive. The activation level of the LA in v^s is set to active. This only active LA upgrades to fire level and selects an action. The selected action corresponds to one of the edges of v^s. This edge is added to the list of edges for the current traverse. The process of adding edges to the current traverse continues until all LAs downgrades to off. At this point, the traverse is completed, that is, a path is formed from v^s to v^d. Rest of the algorithm is the same as Ref. 1.

Misra and Oommen[22] introduced two different versions of an algorithm based on LA to solve the shortest path problem in stochastic graphs. The first version, called LASPA-RR, uses Ramalingam and Reps' scheme[23] in its iterations. The second, called LASPA-FMN, is an algorithm which uses the scheme introduced by Frigioni *et al.*[24] Generally, an LASPA algorithm consists of two steps: initialization and iteration. To begin the initialization step, the algorithm obtains a snapshot of the directed graph with each edge having a random weight. Next, Dijkstra's algorithm[25] is run once to determine the shortest path edges on the graph snapshot. The algorithm maintains an action probability vector for each node of the graph. This vector contains the probability values to choose different actions; each possible outgoing edge corresponds to a probable action that can be selected for calculating the shortest path tree. Based on the shortest path computed using Dijkstra's algorithm, the action probability vector of each node is updated in that the outgoing edge from a node taken as belonging to the shortest path edge, has an increased probability than before the update. In an iteration of LASPA, first, a node is randomly chosen from the current graph and an action of the associated LA is selected. Second, a new sample of weight from the edge related to selected action is determined. Based on this new value, the new shortest path's tree is recalculated using either RR or FMN algorithm. Finally, the action probability vector for the node whose edge was just selected, is updated in that the edge now potentially belonging to the shortest path's tree has more likelihood of being selected than it before the update. The iteration step of LASPA is repeated for many times until the algorithm converges. Another variation of LASPA has been introduced,[26] in which the principles of the Generalized Pursuit (GP) method[4] is used to learn LAs. In Ref. 27, a similar algorithm is presented capable of computing shortest paths for all possible pairs of source-destination in the graph. In this algorithm, the Floyd Warshall's all-pairs static algorithm28 has been used rather than Dijkstra's algorithm to find shortest paths in a snapshot of the stochastic graph and the Demetrescu and Italiano's algorithm,[29] rather than RR and FMN, has been used to recalculate shortest paths.

LA-based algorithms to solve the shortest problem have been successfully used in many real world applications such as finding maximum clique in stochastic graphs,[30] grid resource discovery,[31] link prediction in adaptive web sites,[32] and dynamic channel assignment.[33]

4. Adaptive Stochastic Petri Net Based on LA

In this section, we briefly review the LA-based adaptive Stochastic Petri net (ASPN-LA7) to analyze learning ability of our proposed algorithm. The ASPN-LA is a stochastic Petri net in which conflicts among immediate transitions are resolved using learning automata. Before a review of ASPN-LA, we give a short review of Petri nets and stochastic Petri nets. Note that what follows is standard notation found in the Petri net literature. For more information on PNs, see Refs. 34 and 35.

Definition 4. *Petri net* (PN) is a triple $\{P, T, W\}$, where P is a non-empty finite set of places, T is a non-empty finite set of transitions, and $W: \big((P \times T) \cup (T \times P)\big) \rightarrow \mathbb{N}$ defines the interconnections of both sets of P and T.

For each element x, either a place or a transition, its preset is defined as $\bullet x = \{y \in P \cup T | W(y, x) > 0\}$ and its post-set is defined as $x\bullet = \{y \in P \cup T | W(x, y) > 0\}$. A marking M is $|P|$-vector and $M(i)$is the non-negative number of tokens in place P_i. A transition t is defined as enabled in marking M (denoted by $M[t\rangle$), if for any place $p_i \in \bullet t$, $M(i)$ is equal to or greater than $W(p, t)$. A transition t can fire if $M[t\rangle$. The firing operation, denoted as $M[t\rangle M'$, means that $[M, t\rangle$ and that $M'$is the next marking in the PN evolution.[36] A PN along with an initial marking M_0 creates a PN system. The set of markings reached from M_0 is called a reachability set and represented by a reachability graph.

Definition 5. *Stochastic Petri net* (SPN)[a] is a PN which is defined by a 6-tuple $\{P, T, W, R, \omega, M_0\}$, where

- P, T, W, M_0 are defined as in Definition 4,
- $\forall t \epsilon T, R_t \in \mathbb{R}^+ \cup \{\infty\}$ is the rate of exponential distribution for the firing time of transition t. If $R_t = \infty$, the firing time of t is zero; such a transition is called immediate. On the other hand, a transition t with $R_t < \infty$ is called timed transition. A marking M is said to be vanishing if there is an enabled immediate transition in this marking; otherwise, M is said to be tangible,[37]
- $\forall t \epsilon T, \omega_t \in \mathbb{R}^+$ is the weight assigned to the firing of the enabled transition t, whenever its rate R_t is equal to ∞. The firing probability of transition t enabled in a vanishing marking M is computed as $\omega_t / (\sum_{M[t'\rangle} \omega_{t'})$.

An ASPN-LA is an SPN where immediate transitions are partitioned in some clusters $s_i, i = 1, \ldots, n$ such that the conflicts among enabled transitions in s_i are resolved by an LA7 When ASPN-LA decides to fire an enabled transition from s_i, the associated LA, i.e. LA_i, selects an enabled transition for firing. To partition the immediate transitions into clusters $s_i, i = 1, \ldots, n$, maximal potential conflict set is introduced in Ref. 7 as given in Definition 6. A maximal potential conflict set is referred as a cluster.

[a] It should be noted that the above definition of SPN coincides with the definition of Generalized Stochastic Petri Net (GSPN) given in Ref. 36.

Definition 6. *Maximal Potential Conflict set* is a set of immediate transitions $s_i = \{t_1, t_2, \cdots, t_{n_i}\}$ with following conditions:

- $\{\forall t_k \in s_i, \exists t_j \in s_i | t_k \neq t_j \& \bullet t_k \cap \bullet t_j \neq \emptyset\}$
- $\{\forall t \in T \backslash s_i, \forall t_k \in s_i | \bullet t_k \cap \bullet \mathrm{t} = \emptyset\}$

In ASPN-LA, like in SPN, no special mechanism is used for the resolution of the conflict among timed transitions; the temporal information provides a metric that allows conflict resolution.[36] Therefore, it is necessary to put all timed transitions of ASPN-LA into a single cluster, s_{-1}, in which the temporal information is used to select one enabled timed transition for firing. Formally, an ASPN-LA is defined as follows:

Definition 7. *ASPN-LA* is a 7-tuples $\hat{N} = (\hat{P}, \hat{T}, \widehat{W}, \hat{S}, L, R, \omega^0)$, where

- $\hat{P}$ is a finite set of places.
- $\hat{T} = T \cup T^U$ is a finite set of immediate, timed, and updating transitions. An updating transition $t^u \in T^u$ is an ordinary immediate transition, except that when it fires, the action probability vector of an LA, fused with Petri net, is updated.
- $\widehat{W}: ((\hat{P} \times \hat{T}) \cup (\hat{T} \times \hat{P})) \rightarrow \mathbb{N}$ defines the inter connection of $\hat{P}$ and $\hat{T}$,
- $L = \{LA_1, \dots . LA_n\}$ refers to a set of learning automata with varying number of actions.
- R is defined as in Definition 5.
- $\hat{S} = \{s_{-1}, s_0, s_1, \dots . s_n\}$ denotes a set of non-overlapping clusters, each consisting of a set of transitions:
    - s_{-1} contains all timed transitions in *ASPN-LA*.
    - $s_i, i = 1, \dots, n$ are sets of clusters, each is a maximal potential conflict set. Each cluster s_i is equipped with a learning automaton LA_i. Number of actions of LA_i is equal to the number of transitions in s_i; each action corresponds to one transition.
    - s_0 is the set of remaining transitions in $\hat{T}$.
- $\forall t \epsilon s_0, \omega_t^0 \in \mathbb{R}^+$ describes the weight assigned to the firing of enabled transition t in cluster s_0 .

It is noted that the definition of ω_t in SPN changes to ω_t^0 in ASPN-LA. In other words, in ASPN-LA, weights are only assigned to the immediate transitions in the cluster s_0. This is due to the fact that in ASPN-LA, conflict resolutions among immediate transitions in $s_i, i = 1, \dots, n$ are performed by learning automata, and hence, no weight is required for these set of transitions.

Definition 8. *ASPN-LA system* is a triple $(\hat{N}, M_0, \hat{F})$, where $\hat{N}$ is an APN-LA, M_0 is the initial marking, and $\hat{F} = \{f_1, \cdots, f_n\}$ is the set of reinforcement signal generator functions. $f_\mathrm{i}: M \rightarrow \beta_\mathrm{i}$is the reinforcement signal generator function related to LA_i. A sequence of markings in the APN-LA is the input to f_i and reinforcement signal β_i is its output. Upon the generation of β_i, LA_i updates its action probability vector using the learning algorithm given in Eqs. (1) or (2). For more information on how to construct an ASPN-LA from an SPN and its firing rules an interested reader may refer to Ref. 7.

5. Proposed Algorithm

In the proposed algorithm, we keep traversing all available paths from v^s to v^d within G repeatedly so as to find the shortest path τ^*. To traverse different paths, we use the concept of token. Tokens traverse the paths between $v^s$and v^d, hop by hop, and estimate the time required for traversing different paths. A node v^i within the graph G, receives tokens from all of its neighbors. Assigned to a token received from neighbor v^j is a time which is an estimate of the time required for traversing from v^s to v^i through v^j ($\Gamma_j^{s\sim i}$). Two neighbors v^j and v^k of v^i can be ranked among their estimated times, $\Gamma_j^{s\sim i}$ and $\Gamma_k^{s\sim i}$, that is, if $\Gamma_j^{s\sim i} < \Gamma_k^{s\sim i}$, it means that the path from v^s to v^i which goes through v^j is shorter than the path going through v^k. It is notable since traversing from v^s to v^d is performed repeatedly, the estimates $\Gamma_j^{s\sim i}$ can be changed or even improved over time. The estimates will become stable to some degree when the shortest path between v^s and v^i is the path going through neighbor v^l of v^i for which $\Gamma_l^{s\sim i}$ is the minimum. The question here is that when it is possible to stop traversing different paths and select the shortest path using Γ estimates. The proposed algorithm uses learning automata and its theory to provide a necessary condition which if not met, we still need more traversing.

5.1. *Detailed description of the algorithm*

The proposed algorithm, referred to as VDLA, seeks the shortest path τ^* between v^s and v^d within a stochastic graph. Each node v^i, except for the source node, is equipped with a learning automaton with a variable set of actions LA^i. The number of actions of LA^i is equal to the number of neighbors of v^i; each action is assigned to one neighbor. The probability of selecting each action is initially set to $1/m^i$, where m^i is the number of neighbors of v^i. In addition, each v^i maintains a list $\Gamma^{s\sim i}$, in which, each entry $\Gamma_j^{s\sim i}$ is the estimation of the time required for traversing from v^s to v^i through neighbor v^j. For simplicity in notation, we omit the index s and use Γ^i rather than $\Gamma^{s\sim i}$ hereafter. Γ_j^i is initially set to zero for all is and js.

VDLA algorithm consists of two phases: learning phase and selection phase. In what follows, we will first describe the learning phase, and then give the detailed description of the selection phase.

Learning Phase

Source node v^s initiates the learning phase by starting to repeatedly send out tokens to all of its neighbors according to an exponential distribution with rate λ. These tokens, *Learning-Tokens* (LT), are used to sample lengths of different paths from v^s towards v^d within the graph. Assigned to each LT are an ID number and a timer. When v^s sends out a new learning-token, it creates the token with a new ID number and sets its timer to zero. This newly created LT is then sent out to neighbors of v^s.

Upon the arrival of an LT, sent out by a node v^j, at any neighbor node v^i of v^j, except for v^s, the following steps will be taken:

- Number of LTs arrived at v^i denoted by n^i is increased by one.
- Number of LTs arrived at v^i from v^j denoted by k_j^i is increased by one. Note that $\sum_{j=1}^{m^i} k_j^i = n^i$.
- LA^i is activated and then, the set of all available actions of LA^i, i.e. $\mathcal{A}^i(n^i)$, is created. This set consists of the actions corresponding to the neighbors, from which tokens are arrived to v^i; all of these tokens are waiting to be processed by v^i.
- For all actions in $\mathcal{A}^i(n^i)$, scaled probability vector $\hat{q}^i(n^i)$ is defined as $\hat{q}_k^i(n^i) = \mathrm{q}_k^i(n^i - 1)/K(n^i)$, where $q_k^i(n^i - 1) = prob\left[\alpha^i(n^i - 1) = \alpha_k^i\right]$ and $K(n^i)$ is the sum of the probabilities of actions in subset $\mathcal{A}^i(n^i)$.
- LA^i randomly selects one action from $\mathcal{A}^i(n^i)$ according to $\hat{q}^i(n^i)$. Let the selected action be corresponded to the neighbor v^l.
- Current estimation of the time required for traversing from v^s to v^i through neighbor v^l , Γ_l^i , is updated according to the equation $\Gamma_l^i = (1/k_l^i) \times \left(\left(k_l^i - 1\right) * \Gamma_l^i + LT.timer\right)$.
- The reinforcement signal forLA^i, i.e. β^i, is generated using Eq. (5).

$$\begin{cases} \beta^i = 1;\ LT.timer \geq \frac{1}{m^i} \times \sum\limits_k \Gamma_k^i \\ \beta^i = 0;\ LT.timer < \frac{1}{m^i} \times \sum\limits_k \Gamma_k^i \end{cases} \tag{5}$$

  In other words, if the time required for traversing from v^s to v^i through neighbor v^l is shorter than the average time required for traversing from v^s to v^i through any of the neighbors of v^i, then the selected action of LA^{i} is rewarded (i.e. $\beta^{\mathrm{i}} = 0$). Otherwise, it is penalized (i.e. $\beta^{\mathrm{i}} = 1$).
- Upon the generation of β^i, LA^i updates $\hat{q}^{\mathrm{i}}$ according to Eq. (6) where the learning algorithm L_{R-I} is used.

$$\begin{cases} \hat{q}_k^i(n^i + 1) = \begin{cases} \hat{q}_k^i(n^i) + a\left[1 - \hat{q}_k^i(n^i)\right], & k = l \\ (1 - a)\hat{q}_k^i(n^i), & \forall k \neq l \end{cases};\ \beta^i = 0 \\ \\ \hat{q}^i(n^i + 1) = \hat{q}^i(n^i); \qquad \beta^i = 1 \end{cases} \tag{6}$$

- The probabilities of the actions in subset $\mathcal{A}^i(n^i)$ are rescaled according to $\mathrm{q}_k^i(n^i) = \hat{q}_k^i(n^i + 1) * K(n^i)\ if\ \alpha_k^i \in \mathcal{A}^i(n^i)$.
- If this is the first time that an *LT* with this *ID* number is seen in v^i, then v^i will send out the *LT* to all its neighbors, except for v^l. Repetitive *LT* tokens (tokens with repetitive *ID* numbers) will be discarded to avoid loops.

Selection Phase

In VDLA, v^{d} is responsible to start the selection phase by sending a token, *Selection-Token* (ST), towards v^{s}. This *ST* token is used to find the shortest path τ^*. The *ST* traverses a path τ_k within the graph towards v^{s}. When it arrives at v^{s}, v^{s} stops sending out *LT* tokens. At this time, τ^* is the reverse of the path τ_k.

Upon the arrival of the ST at any node v^{i}, except for v^{s}, the following steps will be taken by v^i:

- It sets the neighbor node v^{j}, from which it receives ST, as its destination node in the shortest path τ^*.
- It sends out ST to the neighbor corresponding to the action with the maximum value of q^i.

The important question here is that when v^{d} can stop the learning phase and start selection phase. One common answer to this question is to set a maximum number of iterations *maxIter* for the learning phase. But using such a condition to terminate the learning phase does not take the learning process into consideration; It is possible that (1) learning does not occur at all at the terminating iteration; (2) learning occurs, but is not matured enough; or (3) learning is completed far sooner than the *maxIter*. In VDLA, we propose a necessary condition on the values of q^i which if it does not hold, the learning has not occurred yet. Thus, using AND operator between this condition and the *maxIter* condition can at least prevent the learning phase to be stopped before any learning occur. This condition can be stated using (7).

$$q_\varepsilon^d(n^d) > LB_\varepsilon^d(n) \tag{7}$$

where

$$LB_\varepsilon^i(n) = \frac{\sum_{k=2}^{m^i}\left[q_k^i(n^i)\Gamma_k^i(n^i)\right]}{\Gamma_\varepsilon^i(n^i)(m^i - 1)} \tag{8}$$

In (7), $\varepsilon = \mathrm{argmin}_j\, \Gamma_j^i(n^i)$. The above condition is achieved when VDLA is modeled by the ASPN-LA to analyze its learning ability (see Sec. 5.2).

Upon receiving an LT token, in addition to the steps explained above, v^{d} takes the following steps:

- If $n^d > maxIter$ Then
- If condition specified by (7) holds then
- Create an ST token and sends it out to the neighbor corresponding to $\mathrm{argmax}_j\, q_j^d(n^d)$.

5.2. *The analysis of the proposed algorithm*

For analysis of learning phase of the proposed algorithm, each node in the graph is modeled by an ASPN-LA7 For this modeling, each node v^i, with $m^i > 0$ edges, can be considered as a queuing system with $m = m^i$ different classes of jobs and one server. A learning-token arriving at a node v^i from a neighboring node v^j is considered as a job from class C_j which arrives into queue Q_j. The service time of a job (learning-token) is assumed to be the required time for that token to traverse from the source node v^s to the node v^i. Two jobs arrive into a queue Q_j have different service times due to the following two reasons: (1) their traversing path from v^s to v^i may differ; and (2) the underlying graph of the problem is stochastic. As a result, the queuing system has unknown parameters, i.e. service times. Since all probability distribution functions describing the statistics of edge lengths of the graph are assumed to be exponential with unknown rate parameters, service time in a queue Q_j is also exponentially distributed with an unknown rate μ_j.[38] The input rates of jobs into queue Q_j is also exponentially distributed with rate $\lambda_j = \lambda, j = 1, .., m$. In other words, the input rates of all queues are equal to λ, which is the rate by which the source node v^s sends out learning-tokens into the graph. By this modeling, finding the shortest path in a stochastic graph is mapped into finding the shortest total waiting time in a queuing system.

We have defined the Priority Assignment (PA) problem in the queuing system with unknown parameters in Ref. 7 as how to select jobs from the queues so that the total waiting time of the system is minimized. The shortest total waiting time of the queuing system with m queues is obtained when the server assigns the highest selection priority to the jobs from the queue with the highest service rate.[39,40] But, if the service rates of queues are unknown, the total waiting time of the system can be shortened by assigning the highest selection probability, rather than priority, to the jobs from the queue with the highest service rate. Using selection probability instead of selection priority is called Probabilistic Priority (PP) mechanism.[7]

The simplest PP mechanism to solve PA problem in a queuing system is to let the server select jobs from any of m queues with equal probability of $\frac{1}{m}$. A better PP mechanism, ASPN-LA-[m]PP, has been introduced in Ref. 7 (Fig. 1). To solve the PA problem by the ASPN-LA-[m]PP, if the class j of jobs is prioritized with a higher probabilistic priority than the class k, then: (1) the LA of ASPN-LA-[m]PP selects jobs from the class j with a higher probability than the class k and (2) the steady-state selection probability of the class j will be higher than that of the class k of jobs. We give a brief description of the ASPN-LA-[m]PP system below.[7]

In the ASPN-LA-[m]PP system shown in Fig. 1, the set $\{t_{2m+k}, k = 1, \dots, m\}$ is a maximal potential conflict and hence, forms a cluster s_1. LA_1 with m actions is assigned to the cluster s_1 to control the conflict among its transitions; each action corresponds to select one transition for firing. The cluster s_{-1} contains all timed transitions and the cluster s_0 contains one transition t_1^u. To obtain the ASPN-LA-[m]PP system, the reinforcement signal generator function set $\hat{F} = \{f_1\}$ is needed. Upon firing of t_1^u, function f_1 is executed which generates the reinforcement signal β_1 for LA_1. To specify how f_1 generates β_1, we first note that when t_1^u fires for the n^{th} time, the following parameters are known to the system[7]:

- The queue from which the last job (job n) was selected for execution;
- The execution time of job n, referred to as $\delta(n)$;
- Number of jobs from Q_j given service so far, referred to as $k_j (\sum_j k_\text{j} = n)$;
- Total service time for the jobs in Q_j given service so far, referred to as $\Delta_j(n)$;
- Average service time for jobs in Q_j given service so far, which can be calculated as $\Gamma_j(n) = \frac{1}{k_j} \times \Delta_j(n)$.

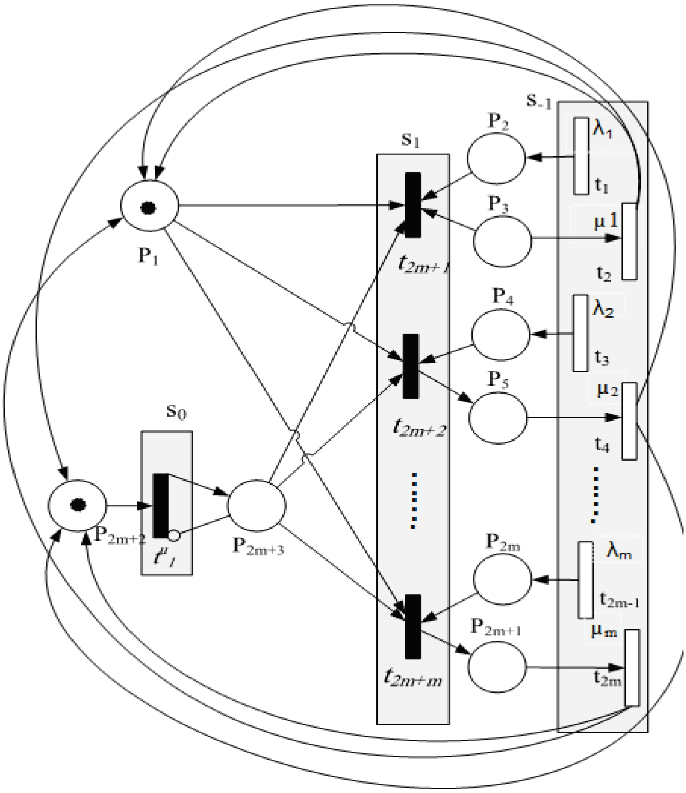

Fig. 1. ASPN-[m]PP model based on ASPN-LA model.

Considering parameters above, f_1 can be described using Eq. (9) where $\beta_1 = 1$ denoted a penalty signal and $\beta_1 = 0$ is considered as a reward signal. Using β_1, generated according to Eq. (9), LA_1 updates its available action probability vector, $\hat{q}(n)$, according to the $L_{\mathrm{R-I}}$ learning algorithm described in Eq. (1). Then the probability vector of the actions of the chosen subset is rescaled according to Eq. (4).

$$\begin{cases} \beta_1 = 1;\ \delta(n) \geq \frac{1}{\mathrm{m}} \times \sum_{\mathrm{k}} \Gamma_{\mathrm{k}}(n), \\ \beta_1 = 0;\ \delta(n) < \frac{1}{\mathrm{m}} \times \sum_{\mathrm{k}} \Gamma_{\mathrm{k}}(n), \end{cases} \tag{9}$$

The Analysis of ASPN-LA-[m]*PP*

In this section, we will show that LA_1 associated with cluster s_1 gradually learns to assign a higher probabilistic probability to the class j of jobs rather than the class k, if $\mu_{\mathrm{j}} > \mu_{\mathrm{k}}$. To better clarify the idea, first, we first breifly review the results of analysis for a simple case of an ASPN-LA-[*2*]PP with two number of queues, i.e. $m = 2$, from Ref. 7. We will then generalize this analysis of the ASPN-LA-[*2*]PP with two queues to the general form of ASPN-LA-[m]PP with m queues. Finally, we will show that LA solves the PA problem in the ASPN-LA-[m]PP system, if L_{R-I} is used as its learning algorithm.

To analyze the ASPN-LA-[*2*]PP in Ref. 7, its underlying Continuous-Time Markov Chain (CTMC) [42] has been utilized. Having derived this CTMC, its states have been portioned into three groups: $\mathcal{S}_1$) a job is selected from Q_1 while another job is waiting in Q_2; $\mathcal{S}_2$) a job is selected from Q_2 while another job is waiting in Q_1; and, $\mathcal{S}_3$) remaining states. Let $\mathcal{P}_i$ denote the summation of the steady-state probability of markings in the sets $\mathcal{S}_i, i = 1,2,3$. Assuming $\mu_1 > \mu_2$, a condition has been introduced in Ref. 7 if hold, ASPN-LA-[*2*]PP assigns selection probabilities $q_j, j = 1,2$ to the queues such that: (1) $q_1 > q_2$ which indicates that a higher probabilistic priority is given to the first class of jobs rather than the second class of jobs and (2) $\mathcal{P}_1 > \mathcal{P}_2$ which indicated that the accumulated sojourn time of $\mathcal{S}_1$ will be longer than that of $\mathcal{S}_2$ in the steady-state. Theorem 1, which has been proven in Ref. 7, provides this condition on $q_i(n),\ i = 1,2$. In Theorem 2, we will generalize Theorem 1 to the ASPN-LA-[m]PP system with m queues. In Theorem 3, we will show that if the L_{R-I} algorithm is used to update the action probability vector of LA_1, then $\mathcal{P}_j > \mathcal{P}_k$ for $j < k$; $j = 1, \dots, m-1$; $k = 1, \dots, m$ when n goes to infinity. To follow CTMC analysis by ordinary methods, the stochastic information attached to the arcs should be fix values. This is why we have assumed fixed values q_1^*, q_2^*, …,q_m^* rather than $q_1(n)$, $q_2(n)$, …,$q_m(n)$, respectively.

Theorem 1. *Let* q_1^* *and* q_2^* *be the selection probabilities of* Q_1 *and* Q_2 *in an ASPN-LA-*[*2*]*PP. If* $\frac{q_1^*}{\mu_1} > \frac{q_2^*}{\mu_2}$, *then the ASPN-LA-*[*2*]*PP assigns a higher probabilistic priority to the first class rather than the second class of jobs.*

Proof. It has been proven in Ref. 7.

Corollary 1. *Assuming* $\mu_1 > \mu_2$, *if selection probability of the first queue passes the lower bound* $\frac{\mu_1 \times q_2^*}{\mu_2}$, *then the shortest total waiting time in the ASPN-LA-*[2]*PP is reached. That is, when the action probability* $q_1(n)$ *is higher than* $\frac{\mu_1 \times q_2(n)}{\mu_2}$, *the shortest total waiting time in the ASPN-LA-*[2]*PP is reached.*

Proof. The proof is immediately followed by the Theorem 1.

Theorem 2. *Let* $q_j^*, j = 1, \dots, m$ *be the selection probabilities of* $Q_j, j = 1, \dots, m$ *in an ASPN-LA-*[m]*PP. If* $q_j^*/\mu_j > q_k^*/\mu_k, 1 \le j,k \le m, j \ne k$, *then the ASPN-LA-*[$m$]PP *assigns a higher probabilistic priority to the* j^{th} *class rather than the* k^{th} *class of jobs, i.e.* $\mathcal{P}_j > \mathcal{P}_k$.

Proof. The proof is given in Appendix A.

Corollary 2. *Suppose that the inequality* (10) *holds for* $j = 1, \dots, m-1$. *Then, the ASPN-LA-*[m]*PP assigns a higher probabilistic priority to the* j^{th} *class rather than the* k^{th} *class of jobs for* $1 \le j,k \le m,\ j < k$.

$$q_j^* > \frac{\mu_j}{(m-j)} \left(\sum_{\ell=j+1}^{m} \frac{q_\ell^*}{\mu_\ell} \right) \tag{10}$$

Proof. By Theorem 2, if there are $m-j$ inequalities for queue j of the form $q_j^*/\mu_j >$, $k = j+1, \dots, m$, then the ASPN-LA-$[m]$PP assigns a higher probabilistic priority to the j^{th} class rather than the k^{th} class of jobs for $1 \le j, k \le m,\ j < k$. Summing up the two sides of these inequalities together we get the inequality (10).

Corollary 3. *Assuming $\mu_1 > \mu_2 > \cdots > \mu_m$, when all inequalities $q_j^* > \left(\mu_j/(m-j)\right) \left(\sum_{\ell=j+1}^{m} q_\ell^*/\mu_\ell\right)$ for $j = 1, \dots, m-1$ hold, the shortest total waiting time in the ASPN-LA-$[m]$PP is reached. That is, when the action probability $q_j(n)$ is higher than $\left(\mu_j/(m-j)\right)\left(\sum_{\ell=j+1}^{m} q_\ell(n)/\mu_\ell\right)$, the shortest total waiting time in the ASPN-LA-$[m]$PP is reached.*

Proof. The proof is immediately followed by the Theorem 2.

Theorem 3. *Let ASPN-LA-$[m]$PP be a queuing system with m queues such that $\mu_1 > \mu_2 > \cdots > \mu_m$. If the learning algorithm L_{R-I} is used to update the action probability vector $q(n)$ of LA_1, then the inequality* (10) *holds for $j = 1$ when n goes to infinity.*

Proof. The proof is given in Appendix B.

Corollary 4. *Let ASPN-LA-$[m]$PP be a queuing system with m queues such that $\mu_1 > \mu_2 > \cdots > \mu_m$ in which the learning algorithm L_{R-I} is used. The ASPN-LA-$[m]$PP assigns the highest probabilistic priority to the first class of jobs.*

Proof. The proof is immediately followed by Theorem 3.

The Proposed Necessary Conditions

In the beginning of this section, we mapped the shortest path problem in stochastic graphs into the priority assignment problem in a queuing system with unknown parameters. Using this mapping, we are now able to use the results of this section to set a necessary condition for termination of the proposed VDLA algorithm. The current estimation of the time required for traversing from v^s to v^i through neighbor v^j, i.e. Γ_j^i, is an estimation of μ_j^{-1}. Without loss of generality, in node v^i, in time instant n, we can suppose that $1/\Gamma_1^i(\mathrm{n}) > 1/\Gamma_2^i(\mathrm{n}) > \cdots > 1/\Gamma_m^i(n)$. By Theorem 2, if

$$q_j^i(n) > LB_j^i(n), j = 1, \dots, m-1 \tag{11}$$

where $LB_j^i(n)$ is defined in Eq. (12), then node v^i significantly prefers the path from v^s to v^i which passes through neighbor v^j to the paths passing through other neighbors. That is, a necessary condition for termination of VDLA algorithm is that in the destination node v^d, the inequalities (11) hold for $i = d$ whereas the selection phase, described in Section 5.1, considers inequalities (11) only for $j = \varepsilon$.

$$LB_j^{(i)}(n) = \frac{\sum_{\ell=j+1}^{m}\left[q_\ell^i(n^i)\Gamma_k^i(n^i)\right]}{\Gamma_j^{(i)}(n^i)(m^i-j)}, j = 1, \dots, m-1 \tag{12}$$

6. Simulation Results

In this section, we conduct a set of computer simulations to study the VDLA performance compared to a DLA-based algorithm[1] and a Particle Swarm Optimization (PSO) approach,[43] in terms of the two criteria of simulation time and number of samplings. To this end, we consider six different graphs constructed form two directed stochastic graphs, Graph A and Graph B, taken from Refs. 1 and 19, shown in Figs. 2 and 3, respectively. Graph A is a directed stochastic graph with 10 nodes, 23 arcs, $v^{\mathrm{s}} = 1$, $v^{\mathrm{d}} = 10$, and $\tau^* = [1, 4, 9, 10]$. Edge cost distribution of Graph A is given in Table . Graph B is a stochastic graph with 15 nodes, 42 arcs, $v^{\mathrm{s}} = 1$, $v^{\mathrm{d}} = 15$, and $\tau^* = [1, 2, 5, 15]$. Edge cost distribution of Graph B is given in Table 2. To evaluate our proposed necessary conditions, we generate three different stochastic graphs based on Graph A, denoted by A(1), A(2), and A(3). These three graphs only differ in the edge cost distributions of the edges involved the shortest path of the Graph A (Table 3). In other words, the probabilistic lengths of the shortest path in these three graphs differ from each other. Similarly, three different stochastic graphs B(1), B(2), and B(3), with different lengths of the shortest path, are generated based on Graph B (Table 5).

In the simulations, LA utilize the L_{R-I} learning algorithm, where *α*, the reward rate, is set to .01. The value of *maxIter* is set to 4000 for graphs A(1), A(2), and A(3) and to 7000 for graphs B(1), B(2), and B(3).

We consider three different versions of the proposed VDLA in the simulations. The first version, denoted by VDLA(1), is similar one described in Section 5.1. In the second version, denoted by VDLA(2), we consider all conditions given by inequalities (11) instead of using condition given by inequality (7) as the necessary condition for the termination of the algorithm. Therefore, in the learning phase of VDLA(2), v^{d} takes the following steps:

- If $n^d > maxIter$, then
    - Define a temporary vector $\bar{\Gamma} = sort\ (\Gamma^d(n^d))$ and construct another temporary vector $\bar{q}$ such that the action probability $\bar{q}_j$ is the probability of the action corresponding to $\bar{\Gamma}_j$.
    - If inequality (13) holds for all $j = 1, \dots, (m^{\mathrm{d}} - 1)$ then
        - Create an *ST* token and sends it out to the neighbor corresponding to the action with the maximum value of q^d.

$$\bar{q}_j > \overline{LB}_j^d(n^i) \tag{13}$$

$$\overline{LB}_j^d(n) = \frac{\sum_{\ell=j+1}^{m}[\bar{q}_k\bar{\Gamma}_k]}{\bar{\Gamma}_j(m^d-j)}, j = 1, \dots, (m^{\mathrm{d}} - 1) \tag{14}$$

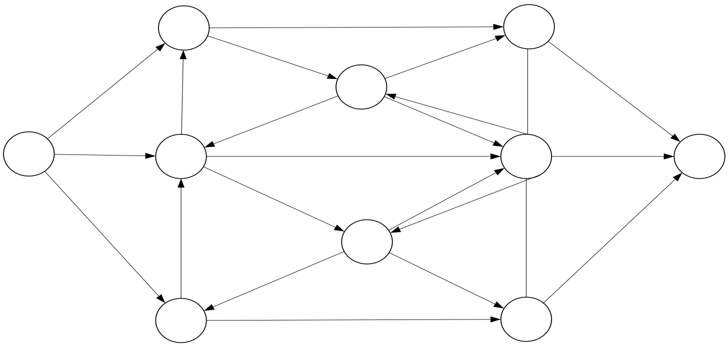

Fig. 2. Graph A.

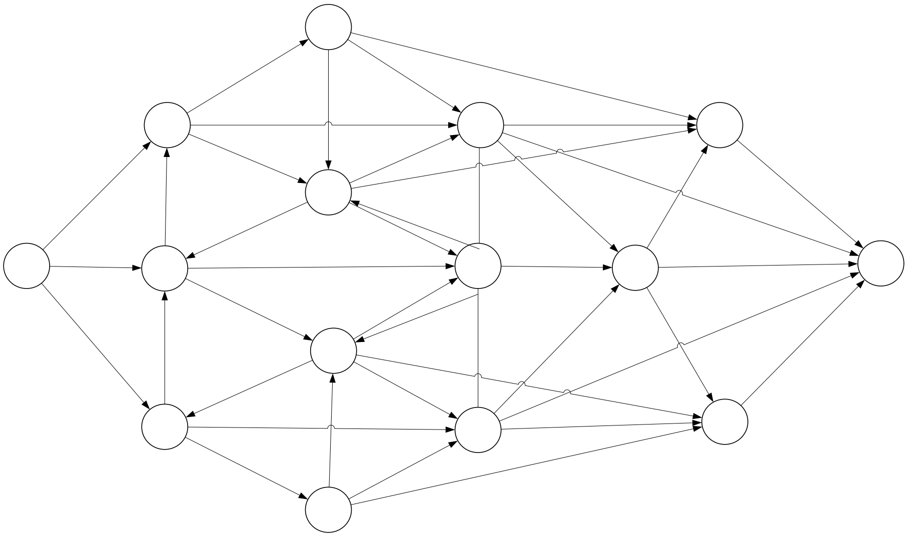

Fig. 3. Graph B.

Table 1. Weight distribution of Graph A (Fig. 2).

Edges	Lengths				Probabilities				Edges	Lengths				Probabilities			
(1,2)	3.0	5.3	7.4	9.4	.2	.2	.3	.2	(6,3)	6.8	7.7	8.5	9.6	.4	.1	.1	.4
(1,3)	3.5	6.2	7.9	8.5	.3	.3	.2	.2	(6,5)	.6	1.5	3.9	5.8	.2	.2	.3	.3
(1,4)	4.2	6.1	6.9	8.9	.2	.3	.2	.3	(6,7)	2.1	4.8	6.6	7.5	.2	.4	.2	.2
(2,5)	2.6	4.1	5.5	9.0	.2	.2	.4	.2	(7,6)	4.1	6.3	8.5	9.7	.2	.3	.4	.1
(2,6)	5.8	7.0	8.5	9.6	.3	.3	.2	.2	(7,8)	1.6	2.8	5.2	6.0	.2	.3	.3	.2
(3,2)	1.5	2.3	3.6	4.5	.2	.2	.3	.3	(7,9)	3.5	4.0	5.0	7.7	.1	.2	.4	.3
(3,7)	6.5	7.2	8.3	9.4	.5	.2	.2	.1	(7,10)	1.6	3.4	8.2	9.3	.2	.3	.3	.2
(3,8)	5.9	7.8	8.6	9.9	.4	.3	.1	.2	(8,4)	7.0	8.0	8.8	9.4	.2	.2	.2	.4
(4,3)	2.1	3.2	4.5	6.8	.2	.2	.3	.3	(8,7)	2.1	4.6	8.5	9.6	.4	.2	.2	.2
(4,9)	1.1	2.2	3.5	4.3	.2	.3	.4	.1	(8,9)	1.7	4.9	6.5	7.8	.2	.2	.2	.4
(5,7)	3.2	4.8	6.7	8.2	.2	.2	.3	.3	(9,10)	4.6	6.4	7.6	8.9	.4	.1	.2	.3
(5,10)	6.3	7.8	8.4	9.1	.2	.2	.4	.2									

Table 2. Weight distribution of Graph B (Fig. 3).

Edges	Lengths				Probabilities				Edges	Lengths				Probabilities			
(1,2)	16	25	36		.6	.3	.1		(7,8)	12	15	22	24	.3	.3	.3	.1
(1,3)	21	24	25	39	.5	.2	.2	.1	(7,10)	19	23	37		.6	.2	.2	
(1,4)	11	13	26		.4	.4	.2		(8,4)	13	23	34		.4	.3	.3	
(2,5)	11	30			.7	.3			(8,7)	14	34	39		.6	.2	.2	
(2,6)	13	37	39		.6	.2	.2		(8,9)	13	31	32		.8	.1	.1	
(2,11)	24	28	31		.5	.3	.2		(8,14)	14	15	27	32	.3	.3	.2	.2
(3,2)	11	20	24		.6	.3	.1		(9,7)	10	17	20		.6	.3	.1	
(3,7)	23	30	34		.4	.3	.3		(9,10)	16	18	36	39	.3	.3	.2	.2
(3,8)	14	23	34		.5	.4	.1		(9,14)	19	24	29		.4	.3	.3	
(4,3)	22	30			.7	.3			(9,15)	12	23	25	32	.4	.3	.2	.1
(4,9)	35	40			.6	.4			(10,13)	14	20	25	32	.3	.3	.2	.2
(4,12)	16	25	37		.5	.4	.1		(10,14)	23	34			.9	.1		
(5,7)	15	17	19	26	.3	.3	.3	.1	(10,15)	15	19	25		.4	.3	.3	
(5,10)	27	33	40		.4	.3	.3		(11,5)	18	19	20	23	.3	.3	.3	.1
(5,13)	28	35	37	40	.4	.3	.2	.1	(11,6)	10	19	39		.5	.4	.1	
(5,15)	25	32			.7	.3			(11,13)	13	31	25		.6	.3	.1	
(6,3)	18	24			.7				(12,8)	15	36	39		.5	.3	.2	
(6,5)	18	25	29		.5	.3	.2		(12,9)	16	22			.7	.3		
(6,7)	11	31	37		.5	.4	.1		(12,14)	10	13	18	34	.3	.3	.3	.1
(6,13)	21	23			.5	.5			(13,15)	12	31			.9	.1		
(7,6)	12	23	31		.5	.3	.2		(14,15)	14	19	32		.5	.3	.2	

Table 3. The edge cost distributions of the edges along τ^* and the probabilistic length of $\tau^*$for A(1), A(2), and A(3). The lengths of edges are shown in Table 1.

	A(1)				**A(2)**				**A(3)**			
Edges of $\boldsymbol{\tau}^*$	Probabilities				Probabilities				Probabilities			
(1,4)	.2	.3	.2	.3	.5	.3	.1	.1	.7	.1	.1	.1
(4,9)	.2	.3	.4	.1	.5	.3	.1	.1	.7	.1	.1	.1
(9,10)	.4	.1	.2	.3	.5	.3	.1	.1	.7	.1	.1	.1
prob. length of $\boldsymbol{\tau}^*$	16.1				13.37				12.41			

Table 4. The edge cost distributions of the edges along τ^* and the probabilistic length of $\tau^*$for B(1), B(2), and B(3). The lengths of edges are shown in Table 5.

	B(1)			**B(2)**			**B(3)**		
Edges of $\boldsymbol{\tau}^*$	Probabilities			Probabilities			Probabilities		
(1,2)	.6	.3	.1	.8	.1	.1	.9	.1	.0
(2,5)	.7	.3		.8	.2		.9	.1	
(5,15)	.7	.3		.8	.2		.9	.1	
prob. length of $\boldsymbol{\tau}^*$	64.5			60.1			55.5		

In the last version, denoted by VDLA(3), not only the destination node v^{d}, but also any node receiving an ST token, except for v^{s}, checks the condition given by the inequality (7) and sends the ST towards v^{s} only after this condition holds. In VDLA(3), upon the arrival of the ST at any node v^{i}, except for v^{s}, the following two steps will be taken by v^{i}:

- First, it sets the neighbor node v^{j}, from which it receives ST, as its destination node in the shortest path τ^*.

- Second, it sends out the ST to the neighbor corresponding to the action with the maximum value of q^i if Eq. (15)(15) hold where $LB_\varepsilon^i(n^i)$ is defined in Eq. (8) for $= \text{argmin}_j\, \Gamma_j^i(n^i)$.

$$q_\varepsilon^i(n^i) > LB_\varepsilon^i(n^i) \tag{15}$$

6.1. *Experiment one*

This experiment is conducted to study the ability of the proposed algorithm in finding the shortest path in different graphs. We define n^* as the iteration number, at which the proposed necessary conditions hold for the first time. We also define ℕ as the number of iterations, after n^*, at which the proposed algorithm cannot find the shortest path τ^*. An iteration here is defined as the number of unique LTs arrived at the destination node v^d.

All reported results are averaged over 50 independent runs of simulation. In each run, v^s repeatedly sends out learning-tokens to all its neighbors according to an exponential distribution with rate $\lambda = .1$. Results for Experiment One are reported in Table 5.

Table 5. Results of simulations on graphs.

Alg.	***G***	$\boldsymbol{n^*}$	ℕ	$\boldsymbol{LB_\varepsilon^d(n^*)}$, Eq. (8)
VDLA(1)	A(1)	571	0	.3500
	A(2)	748	0	.3821
	A(3)	808	0	.3960
	B(1)	488	0	.2051
	B(2)	494	0	.2089
	B(3)	499	0	.2076
Alg.	Graph	n^*	ℕ	$\overline{LB}_j^d(n^*), j = 1, \dots, m^d$, Eq. (14)
VDLA(2)	A(1)	624	0	[.3499, .2894]
	A(2)	739	0	[.3819, .2554]
	A(3)	820	0	[.3961, .2487]
	B(1)	489	0	[.2049,.1764,.1459,.0800]
	B(2)	491	0	[.2088,.1709,.1414,.0763]
	B(3)	497	0	[.2075,.1580,.1218,.0792]
Alg.	Graph	n^*	ℕ	$LB_\varepsilon^i(n^*), v^i\ along\ \tau^*$, Eq. (8)
VDLA(3)	A(1)	2319	0	.9558, .4837, .3499
	A(2)	2789	0	.9941, .5882, .3820
	A(3)	2954	0	.9952, .6060, .3961
	B(1)	4782	0	.6399, .4542, .2050
	B(2)	4983	0	.6858, .4838, .2089
	B(3)	4983	0	.6972, .4997, .2074

From Table 5, one may conclude the following remarks:

- The shorter the length of the shortest path, the higher the values of $LB^d(n^*)$ and n^* will be. To account for this phenomenon, recall that Γ_ε^d is an estimate of the length of the shortest path in the graph in Eq. (8). Considering the Eq. (8), a decline in the value of Γ_ε^d results in a rise in the value of LB_ε^d. As a result, when the length of the shortest path in the graph decreases, Γ_ε^d decreases and subsequently LB_ε^d increases. When LB_ε^d increases, the number of tokens required by LA^d residing in v^d to pass that LB_ε^d increases, and hence n^* will increase as well.
- The value of n^* in VDLA(3) is higher than that of VDLA(1) and VDLA(2). This is due to the fact that in VDLA(3), the algorithm terminates condition given by Eq. (15) must hold in all nodes v^i along the shortest path.

In all simulations, for time instant $n > n^*$, the path constructed by the VDLA is equal to the shortest path τ^*. In other words, although the proposed termination condition is proved to be a necessary condition, our experiments suggest that it may be possible to consider it as a sufficient condition for the termination of the algorithm.

6.2. *Experiment two*

We conduct a set of simulations to compare VDLA(1) and a DLA-based algorithm, introduced in Ref. 1, to solve the shortest path problem. To compare the results, three performance measures are calculated: (1) the average number of sampling taken from the edges of graph, denoted by AS, (2) the average required time for all traversing into graph, denoted by AT, and (3) the average required time for taking a sample from the edges of graph, which can be calculated by the division of AT to AS denoted by ATS ($\mathrm{ATS} = \mathrm{AT}/\mathrm{AS}$). All reported results are averaged over 50 independent runs of simulations. In each run, we calculate the measures until the number of updates in LA^{d} reaches a specific number. In Table 6, this number is reported in a column with "Number of Updating". For example, in A(1), to update LA^d for 1000 times, the DLA algorithm takes 6351 samples from the graph in 37602 milliseconds whereas VDLA(1) takes 23024 samples in 9638 milliseconds.

From Table 6, following points can be concluded:

- The average number of samplings for DLA algorithm is significantly lower than that of VDLA(1).
- The average required time for all traversing into the graph for VDLA(1) is substantially shorter than that of DLA algorithm. In other words, VDLA(1) takes more samples from the graph in lower time than DLA does.
- The average required time for taking a sample from edges in the graph for VDLA(1) is significantly shorter than that of DLA algorithm. In other words, the speed of taking a sample from the graph in VDLA(1) is higher than that of DLA.

Table 6. Results of simulations on graphs A(1), A(2), and A(3).

		DLA			**VDLA(1)**		
$\boldsymbol{G}$	Number of Updating	AS	AT(ms)	ATS(ms)	AS	AT(ms)	ATS(ms)
A(1)	1000	6351	37602	5.92	23024	9638	.42
	2000	12764	75439	5.91	46029	19927	.43
	3000	18786	111035	5.91	69023	30347	.44
A(2)	1000	6549	36972	5.65	23024	10083	.44
	2000	12858	72808	5.66	46043	19949	.43
	3000	18752	104465	5.57	69016	29782	.43
A(3)	1000	6291	34803	5.53	23003	10188	.44
	2000	12559	69543	5.54	46003	18854	.41
	3000	18646	100647	5.40	69025	29948	.43
B(1)	1000	7840	164870	21.03	23519	5518	.23
	2000	15586	328710	21.10	47105	11320	.24
	3000	23261	489229	21.03	70408	16831	.24
B(2)	1000	7900	163788	20.73	23551	5451	.23
	2000	15309	315755	20.63	47013	11352	.24
	3000	23512	485602	20.65	70474	17615	.25
B(3)	1000	7628	155027	20.32	23565	5621	.24
	2000	15392	312617	20.31	47010	11352	.24
	3000	23136	473529	20.47	70369	16284	.23

6.3. *Experiment three*

We conduct a set of simulations to compare VDLA(1) and a Particle Swarm Optimization (PSO) approach, introduced in Ref. 43. To compare the results, the percentage of finding the shortest path (referred to as FSP) and the value of ATS, i.e. the average required time for taking a sample from edges of the graph (described in Experiment Two) are calculated. All reported results are averaged over 50 independent runs of simulations. Each simulation is terminated when the iteration number is reached to the maximum number of iterations. In Table 7, this maximum number is reported in a column titled "Max. Number of Iterations". At the end of a simulation, a path is determined by the algorithm and then, FSP measure is updated using this path. Like experiment One, in VDLA(1) algorithm, an iteration is defined as the number of unique LTs arrived at the destination node v^d. Similarly, for PSO algorithm, an iteration is defined as the number of particles arrived at v^d.

Table 7. Comparisons of VDLA(1) and PSO approach introduced in Ref. 43.

G	Max. Iterations	Finding the Shortest Path (FSP)			ATS		
		PSO-50	PSO-100	VDLA(1)	PSO-50	PSO-100	VDLA(1)
A(1)	1000	97	98	100	5.37	4.82	.42
	2000	99	100	100	5.36	4.81	.43
	3000	100	100	100	5.36	4.82	.44
A(2)	1000	95	96	100	5.13	4.61	.44
	2000	96	98	100	5.14	4.61	.43
	3000	98	99	100	5.06	4.54	.43
A(3)	1000	95	96	100	5.02	4.51	.44
	2000	95	98	100	5.03	4.51	.41
	3000	97	99	100	4.90	4.41	.43
B(1)	1000	68	63	100	18.95	16.87	.23
	2000	78	75	100	19.01	16.93	.24
	3000	84	80	100	18.95	16.87	.24
B(2)	1000	67	63	100	18.68	16.63	.23
	2000	76	74	100	18.59	16.55	.24
	3000	85	80	100	18.61	16.57	.25
B(3)	1000	63	60	100	18.31	16.30	.24
	2000	75	69	100	18.30	16.30	.24
	3000	82	76	100	18.45	16.42	.23

From Table 7, following points can be concluded:

- VDLA(1) always outperforms the PSO algorithm in finding the shortest path. For example, in the graph B(3), VDLA(1) finds the shortest path before the 1000th iteration, whereas, even in the 3000th iteration, PSO with 50 particles and PSO with 100 particles are only able to find the shortest path in 82 and 76 percentage of times, respectively.
- Considering the fact that the average required time for taking a sample from edges of the graph in VDLA(1) is significantly lower than that of PSO, one may conclude that using VDLA(1), speeds up the process of finding the shortest path in stochastic graphs to a noticeable extent in comparison to PSO algorithm.

Since the VDLA takes samples from all edges of the graph, we can extend VDLA algorithm to solve the single-source shortest path problem. We propose an extension of VDLA to solve this problem in stochastic graphs below.

6.4. *Algorithm extension: the single-source shortest path*

Here, we extend our proposed algorithm to find the shortest paths from the source node v^{s} to all other nodes known as the single-source shortest path problem.[44] In the extended algorithm, denoted by eVDLA, each node v^{i}, except for v^{s}, is responsible for starting the selection phase by sending a selection-token, denoted by ST^{i}, to v^{s}. ST^{i} is used to find the shortest path from the v^{s} to v^{i} denoted by $\tau^{*\mathrm{i}}$. ST^{i} traverses this path within the graph towards v^{s}. When v^{s} gets all ST^{i}, v^{s} stops sending out LT tokens.

Upon the arrival of ST^{j} at any node v^{i}, except for v^{s}, the following steps will be taken by v^{i}:

- It sets the neighbor node v^{k}, from which it receives ST^{j}, as its next node in the shortest path τ^{*j}.
- It sends out ST^{j} to the neighbor corresponding to the action with the maximum value of q^{i}.

Table 8. Results of different simulation by eVDLA on graphs A(1) and B(1).

Graph	A(1)	B(1)
Destination	n^*	n^*
v^2	1417	4782
v^3	2493	6417
v^4	2319	5982
v^5	1163	1872
v^6	889	2124
v^7	1172	1723
v^8	769	2086
v^9	1185	1850
v^{10}	571	936
v^{11}	–	3982
v^{12}	–	3879
v^{13}	–	1018
v^{14}	–	975
v^{15}	–	488

A set of computer simulations is conducted to study the proposed necessary condition over all nodes of graphs. To do this, we consider the inequality (15) as the necessary condition in eVDLA and report the average value of n^* for graphs A(1) and B(1) in Table 8, which summarizes the results for 50 different simulations.

In all simulations, for time instant $n > n^*$, the path constructed by the VDLA for v^i is equal to the shortest path τ^{*i}. In other words, although the proposed termination condition is proved to be a necessary condition, our experiments suggest that it may be possible to consider it as a sufficient condition for the termination of the algorithm.

6.5. *Algorithm improvement: using discrete learning automata*

There is always a need to improve the speed of the operation of a learning automaton.[45] One of the way in which such an improvement can be fulfilled is discretizing the action probability space of the LA.[45] In discretized automata models we restrict the action probabilities to a finite number of values in the interval [0,1]. The number of such values denotes the level of discretization and is a design parameter. The values are generally spaced equally in [0,1]. Every linear learning algorithm considered by a variable structure LA, can be discretized in this manner.[45]

In this experiment, we use a discretized L_{RI} algorithm to update the action probability vectors of LAs in VDLA. In the improved version of VDLA, denoted by dVDLA, the following learning algorithm is used instead of the ordinary L_{RI} algorithm. Let $\mathcal{N}$ be the resolution parameter indicating the level of discretization. The smallest change in any action probability is then chosen as $\Delta = \frac{1}{r\mathcal{N}}$ where r is the number of actions. Let $\alpha(n) = \alpha_i$ be the action chosen by the learning automaton at instant n

$$q_j(n+1) = \begin{cases} \max\{q_j(n) - \Delta, 0\}, \forall j \neq i \\ 1 - \sum_{j \neq i} q_j(n) \qquad , j = i \end{cases} \tag{16}$$

when the taken action is rewarded by the environment (i.e. $\beta(n) = 0$) and

$$q_j(n+1) = q_j(n), \forall j \tag{17}$$

when the taken action is penalized by the environment (i.e. $\beta(n) = 1$).

Utilizing discretized LAs, instead of ordinary LAs, the newly versions of VDLA(1), VDLA(2), and VDLA(3) are denoted by dVDLA(1), dVDLA(2), and dVDLA(3) respectively. Simulation settings of this experiment are completely identical to that used in Experiment one. The resolution parameter $\mathcal{N}$ is set to 100. The results for this experiment are reported in Table 9. As it was anticipated, using discretized learning automata increases the speed of the algorithm by a factor of 31% on average.

Table 9. Results of simulations on graphs.

	$\boldsymbol{n}^*$					
G	VDLA(1)	dVDLA(1)	VDLA(2)	dVDLA(2)	VDLA(3)	dVDLA(3)
A(1)	571	377	624	424	2319	1531
A(2)	748	524	739	488	2789	1952
A(3)	808	566	820	549	2954	2038
B(1)	488	317	489	333	4782	3491
B(2)	494	341	491	354	4983	3438
B(3)	499	344	497	353	4983	3239

7. Conclusion

In this paper, different versions of algorithm VDLA is proposed based on the learning automata to solve the shortest path problem in stochastic graphs. In the proposed algorithm, an LA, assigned to a node, selects a neighbor on the shortest path from the source node to that node. To learn the LAs in VDLA, a number of tokens traverse into the graph and estimate the required time for traversing from different paths. Using these estimations, in the learning phase of VDLA, the LA in each node gradually learns to assign the higher probability to the action related to neighbor along the shortest path. The destination node terminates the learning phase of VDLA by sending a special token, selection-token, toward the source node. To determine when this token is sent, we proposed a number of necessary conditions, if hold, the destination node can send this token to a neighbor along the shortest path. We used a recently introduced model, Adaptive Stochastic Petri Net (ASPN-LA) to find these conditions. We argued that VDLA can assign the highest steady-state probability to select the shortest path if the proposed necessary conditions hold. Computer simulations reported in the paper did in fact verify the theoretical results and compared the experimental results with a DLA-based algorithm. Finally, we proposed eVDLA algorithm to solve the single-source shortest path problem in stochastic graphs.

Acknowledgment

This research was in part supported by a grant from IPM. (No. CS1396-4-21).

Appendix A. Proof of Theorem 2

We prove this property by contradiction. Suppose that $\left(q_j^*/\mu_j\right) \leq (q_k^*/\mu_k)$. Since $\left(\mu_k/\mu_j\right) < 1$ and $q_j^* \times \left(\mu_k/\mu_j\right) < q_k^*$, the relation $q_j^* < q_k^*$ must hold true, which is a contradiction. Due to selection of the class j of jobs with a higher probabilistic priority than the class k, q_j^* must be absolutely greater than q_k^*. Therefore, the Theorem 2 is proved.

Appendix B. Proof of Theorem 3

The shortest total waiting time is achieved, when the highest probabilistic priority is assigned to class of jobs with the shortest average service time.[39,40] From learning procedure L_{R-I} shown in Eq. (1), if action 1 is attempted at instant n, the probability $q_1(n)$ is increased at instant $n+1$ by an amount proportional to $1 - q_1(n)$ for a favorable response and fixed for an unfavorable response. By this, it follows that $\{q(n)\}_{n>0}$ can be described by a Markov-process whose state space is the unit interval [0, 1], when automaton operates in an environment with penalty probabilities $\{c_1, c_2, \dots, c_m\}$. The schema L_{R-I} consists of m absorbing states: $\{e_i(j) = 0 \; and \; e_i(i) = 1\}, i, j = 1, \dots, m$. Since the probability $q_i(n)$ can be decreased only when α_i is chosen and results in a favorable response, the probability $q(k) = e_i$ holds if $q(n) = e_i, i = 1, \dots, m$ for all $k \geq n$. Thus, $V \triangleq \{e_1, e_2, \dots, e_m\}$ represents the set of all absorbing states and the Markov process $\{q(n)\}_{n>0}$ generated by the schema L_{R-I} converges to the set V with probability one.

To study the asymptotic behavior of the process $\{q(n)\}_{n>0}$, a common method is to compute the conditional expectation of $q_1(n+1)$ given $q_1(n)$. For the schema L_{R-I}, this computation shows that the expected value of $q_i(n)$ increases or decreases monotonically with n depending on the values of $c_i, i = 1, \dots, m$. Study on the asymptotic behavior shows that $q_i(n)$ converges to *0* with a higher probability when the value of c_i is the largest value among $c_j, j = 1, \dots, m, j \neq i$ and to 1 with a higher probability when the value of c_i is the lowest value among $c_j, j = 1, \dots, m, j \neq i$ if the initial probability is $q_i(0) = 1/m$ (Ref. 22).

To prove Theorem 3, it is enough to show that penalty signal generated by ASPN-LA-[m]PP system for the first class of jobs, is the lowest. In our queuing system, each class of job has a service time distribution with the exponential density function as Eq. (18).

$$f_i(t) = \mu_i e^{-\mu_i t} \tag{18}$$

The reinforcement signal generator function produces the penalty signal $\beta_i(n) = 1$ for class i in time instant n by probability value $c_i(n), i = 1, \dots, m$. Let $\Gamma(n) = \frac{1}{m} \times \sum_{i=1}^{m} \Gamma_i(n)$ is the average of system waiting time to the instant n. Therefore, $c_i(n)$ is defined as the probability of the service time of n^{th} job will exceed $\Gamma(n)$ and is defined by Eq. (19).[39]

$$c_i(n) = prob[\delta_i(n) > \Gamma(n)] = e^{-\mu_i \Gamma(n)}, i = 1, \dots, m \tag{19}$$

In Eq. (19), $\delta_{\mathrm{i}}(\mathrm{n})$ is the execution time of job n which is selected from i^{th} class for service. Based on our assumption $\mu_1 > \mu_2 > \cdots > \mu_{\mathrm{m}}$, it is clear that $c_1(n) < c_2(n) < \cdots < c_{\mathrm{m}}(n)$ with probability one for all n.[39] Therefore, in ASPN-LA-[m]PP system, $q_1(n)$ converges to 1 with probability one when n goes to infinity. Therefore, the Theorem 3 is proved.

References

1. H. Beigy and M. R. Meybodi, Utilizing distributed learning automata to solve stochastic shortest path problem, *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems* **14** (2006) 591–617.
2. M. R. Mollakhalili Meybodi and M. R. Meybodi, Extended distributed learning automata: An automata-based framework for solving stochastic graph optimization problems, *Applied Intelligence* **41** (2014) 923–940.
3. M. L. Tsetlin, On the behaviour of finite automata in random media, *Automation and Remote Control* **22** (1962) 1210–1219, Originally in *Avtomatika i Telemekhanika* **22** (1961) 1345–1354.
4. K. S. Narendra and M. A. Thathachar, *Learning Automata* (Prentice-Hall: Englewood Cliffs, NJ, 1989).
5. H. Beigy and M. R. Meybodi, A mathematical framework for cellular learning automata, *Advances on Complex Systems* **7** (2004) 295–320.
6. M. Esnaashari and M. R. Meybodi, Irregular cellular learning automata, *IEEE Transactions on Cybernetics* **45** (2014) 1622–1632.

7. S. M. Vahidipour, M. R. Meybodi and M. Esnaashari, Learning automata based adaptive petri net and its application to priority assignment in queuing systems with unknown parameters, *IEEE Transactions on Systems, Man, and Cybernetics* **45** (2015) 1373–1384.
8. A. Rezvanian *et al.,* Sampling from complex networks using distributed learning automata, *Physica A: Statistical Mechanics and Its Applications* **396** (2014) 224–234.
9. J. A. Torkestani and M. R. Meybodi, A learning automata-based cognitive radio for clustered wireless ad-hoc networks, *Journal of Network and Systems Management* **19** (2010) 278–297.
10. M. A. Thathacher and B. R. Harita, Learning automata with changing number of actions, *IEEE Transactions on Systems, Man and Cybernetics* **17** (1987) 1095–1100.
11. M. A. L. Thathachar and P. S. Satstry, A hierarchical system of learning automata that can learn the globally optimal path, *Information Sciences* **42** (1997) 743–166.
12. R. J. Williams, *Toward a Theory of Reinforcement Learning Connectionist Systems*, Technical Report (Northeastern University, Boston, 1988).
13. E. A. Billard, Stability of adaptive search in multi-level games under delayed information, *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* **26** (1996) 231–240.
14. E. A. Billard, Chaotic behavior of learning automata in multi-level games under delayed information, in *Proc. IEEE Intl. Conf. Systems, Man, and Cybernetics* (1997), pp. 1412–1417.
15. M. R. Meybodi, H. Beigy, and M. Taherkhani, Cellular learning automata and its applications, *Sharif Journal of Science and Technology* **19** (2003) 54–77.
16. R. Rastegar, A. R. Arasteh, A. Harriri and M. R. Meybodi, A fuzzy clustering algorithm using cellular learning automata based evolutionary algorithm, in *Proc. Fourth Intl. Conf. Hybrid Intelligent Systems* (*HIS04*) (Japan, Kitakyushu, 2004), pp. 310–314.
17. R. Vafashoar, M. R. Meybodi and A. H. Momeni, CLA-DE: A hybrid model based on cellular learning automata for numerical optimization, *Journal of Applied Intelligence* **36** (Springer Verlag, 2012) 735–748.
18. B. Hashemi and M. R. Meybodi, Cellular PSO: a PSO for dynamic environments, *Advances in Computation and Intelligence*, Lecture Notes in Computer Science **5821** (2009) 422–433.
19. H. Beigy and M. R. Meybodi, A new distributed learning automata based algorithm for solving stochastic shortest path problem, in *Proc. Sixth Int'l Joint Conference on Information Science* (Triangle Park, Durham, NC, USA, 2002) pp. 339–343.
20. M. R. Meybodi and H. Beigy, Solving stochastic path problem using distributed learning automata, in *Proc. Sixth Annual Int. CSI Computer Conference* (Iran, 2001), pp. 70–86.
21. M. R. Meybodi and H. Beigy, Solving stochastic shortest path problem using Monte Carlo sampling method: A distributed learning automata approach, *Springer-Verlag Lecture Notes in Advances in Soft Computing: Neural Networks and Soft Computing* (2003) , pp. 626–632.
22. S. Misra and B. J. Oommen, Stochastic learning automata-based dynamic algorithms for the single-source shortest path problem, in *Proc. Int'l Conf. Industrial and Eng. Applications of Artificial Intelligence and Expert Systems* (2003), pp. 239–248.
23. G. Ramalingam and T. Reps, On the computational complexity of dynamic graph problems, *Theoret. Comput. Sci.* **158** (1996) 233–277.
24. D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, Fully dynamic algorithms for maintaining shortest paths trees, *Journal of Algorithms* **34** (2000) 251–281.
25. E. W. Dijkstra, A note on Two Problems in Connection with Graphs, *Numerische Mathematik* **1** (1959) 269–271.
26. S. Misra and B. J. Oommen, GPSPA: A new adaptive algorithm for maintaining shortest path routing trees in stochastic networks, *International Journal of Communication Systems* **17** (2004) 963–984.
27. S. Misra and B. J. Oommen, An efficient dynamic algorithm for maintaining all-pairs shortest paths in stochastic networks, *IEEE Transactions on Computers* **55** (2006) 686–702.

28. R. W. Floyd, Algorithm 97 (SHORTEST PATH), *Comm. ACM* **5** (1962) 345.
29. C. Demetrescu and G. F. Italiano, A new approach to dynamic all pairs shortest paths, in *Proc. 35th Ann. ACM Symp. Theory of Computing* (2003) pp. 159–166.
30. A. Rezvanian and M. R. Meybodi, Finding maximum clique in stochastic graphs using distributed learning automata, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **23** (2015) 1–31.
31. M. Hasanzadeh and M. R. Meybodi, Grid resource discovery based on distributed learning automata, *Journal of Computing in Springer* **96** (2014) 909–922.
32. M. Mollakhalili Meybodi and M. Meybodi, Link prediction in adaptive web sites using distributed learning automata, *13th Annual CSI Computer Conference of Iran* (Kish Island, 2008).
33. H. Beigy and M. R. Meybodi, Cellular learning automata based dynamic channel assignment algorithms, *International Journal of Computational Intelligence and Applications* (*IJCIA*) **8** (World Scientific, 2009) 287–314.
34. W. Reisig, *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies* (Springer Science & Business, 2013).
35. T. Murata, Petri nets: properties, analysis and applications, *Proc. IEEE* **77** (1989) 541–580.
36. A. Marsan *et al.*, *Modeling with Generalized Stochastic Petri Net*, Wiley Series in Parallel Computing (John Wiley and Sons, 1995).
37. G. Ciardo *et al.*, Automated generation and analysis of Markov reward models using stochastic reward nets, *Linear Algebra, Markov Chains, and Queuing Models* (Springer, New York, 1993), pp. 145–191.
38. J. Freiheit and J. Billington, New developments in closed-form computation for GSPN aggregation, *The 5th Int. Conference on Formal Engineering Methods* (Springer, Berlin Heidelberg, 2003) pp. 471–490.
39. M. R. Meybodi and S. Lashmivarahan, A learning approach to priority assignment in a two class M/M/1 queuing system with unknown parameters, *The Yale Workshop on Adaptive System Theory* (1983), pp. 106–109.
40. A. Cobham, Priority assignment in waiting line problems, *Operations Research* **2** (1954) 70–76.
41. Y. Jiang, C. K. Tham and C. C. Ko, A probabilistic priority scheduling discipline for multi-service networks, *Computer Communications* **25** (2002) 1243–1254.
42. G. Bolch *et al.*, *Queuing System and Markov Chain,* 2nd edn. (Wiley Publication, 2006).
43. S. Momtazi, S. Kafi, and H. Beigy, Solving stochastic path problem: Particle swarm optimization approach, *21st Int'l Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems* (Poland, June 18–20, 2008), pp. 590–600.
44. B. V. Cherkassky, A. V. Goldberg and T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, *Mathematical Programming* **73** (1996) 129–174.
45. M. A. Thathachar and P. S. Sastry, *Networks of Learning Automata: Techniques for Online Stochastic Optimization* (Springer Science & Business Media, 2011).