Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 1489 – 1503

doi:10.3906/elk-1811-46

Dynamic Physarum Solver: a bio-inspired shortest path method of dynamically changing graphs

Hilal ARSLAN*
Roketsan Missiles Inc., Ankara, Turkey

Received: 07.11.2018 • **Accepted/Published Online:** 28.01.2019 • **Final Version:** 22.03.2019

Abstract: In dynamic graphs, edge weights of the graph change with time and solving the shortest path problem in such graphs is an important real-world problem. The studies in the literature require excessive computational time for computing the dynamic shortest path since determining changing edge weights is difficult especially when the graph size becomes large. In this paper, we propose a dynamic bio-inspired algorithm for finding the dynamic shortest path for large graphs based on Physarum Solver, which is a shortest path algorithm for static graphs. The proposed method is evaluated using three different large graph models representing diverse real-life applications. The effect of changing edge weights on the solution time is evaluated for each graph model separately and compared against Δ-stepping, which is the most representative implementation of Dijkstra's algorithm. Experimental results show that the proposed method easily adapts edge weight changes and computes the dynamic shortest path efficiently.

Key words: Physarum Solver, dynamic shortest path, Dijkstra's algorithm, solution of linear systems, M-matrix

1. Introduction

The shortest path problems can be investigated in two categories, which are static and dynamic. While edge weights of the graph do not change with time in the static shortest path problems, in dynamic shortest path problems, the edge weights of the graph may increase, decrease, or both with time. Solving dynamic shortest path problems is time-consuming when compared to static ones. Next, we formally define the shortest path problem for static and dynamic graphs.

Given a graph $G(V, E, w)$, V is the set of vertices, E is the set of edges, and w is a weight function such that E to R^+. When a source and destination vertices are given, the smallest weight path from one vertex to another is determined as the single source–single target shortest path problem. If edge weights of the graph G are changing over time, another graph $G'(V, E, w')$ is obtained such that G' and G have the same set of vertices and edges. The main goal is to compute the shortest path on G' efficiently by using the shortest path information obtained from G. The classical static shortest path algorithms such as Dijkstra's algorithm can be used repeatedly to compute the shortest path whenever the graph changes to solve the dynamic shortest path problems. Those algorithms can be preferable when the number of changing edges is large. Otherwise, it becomes inefficient. When the percentages of changing edges are small, dynamic algorithms are used to solve the dynamic shortest path problems efficiently in order to minimize the shortest path's recomputation time.

The real-world applications are dynamically changing, and to cope with such problems, many algorithms inspired by nature have emerged, such as the ant colony algorithm, genetic algorithm, and Physarum Solver.

*Correspondence: hkilic@ceng.metu.edu.tr

Physarum Solver, which is an iterative amoeba algorithm, is able to solve shortest path problems. To solve dynamic shortest path problems efficiently, in this study, dynamic Physarum Solver is presented. Our study focuses on the solution of linear systems, which is required in each iteration of Physarum Solver. In earlier studies, a direct solver was mostly used in this step, and it computes the solution with high accuracy that is usually not needed. The results are evaluated using three different large dynamic sparse graph models representing diverse real-life applications.

This paper is organized as follows: Section 2 discusses related work. The mathematical models of Physarum and the Δ-stepping algorithm are presented in Section 3. Section 4 explains the proposed bio-inspired method for solving the dynamic shortest path efficiently. Section 5 presents experimental results, and Section 6 includes conclusions.

2. Related work

In dynamic shortest path problems, the edge weights can change in three different ways: edge weights may increase, decrease, or both. The algorithms that are designed for graphs whose edge weights either strictly increase or decrease are called semidynamic. Moreover, the algorithms that permit mixed edge weight changes are called fully dynamic. Therefore, dynamic algorithms are further divided into two categories, which are semidynamic and fully dynamic. These algorithms compute the shortest path by recomputing only affected vertices and so they save time when a small portion of the edge weights change.

King [1] proposed the first fully dynamic algorithm to solve all pairs shortest path problems on directed graphs with positive edge weights. Then Demetrescu and Italiano [2] introduced the first fully dynamic algorithm on directed graphs with real edge weights by improving King's algorithm. Narvaez et al. [3] introduced a new dynamic algorithm that makes use of the previously computed shortest path information in the graph to overcome single edge weigh change [4]. To update changes more efficiently, Narvaez et al. [5] proposed another algorithm called BallString [5], which is a semidynamic algorithm. They compared their results with existing results and concluded that BallString is the best-performing algorithm when the changes are small. Chan and Yang [6] proposed the DynDijkstra algorithm, which is a semidynamic version of the Dijkstra algorithm. Ramalingam and Reps [7] proposed a fully dynamic version of the algorithm, which is called Dynamic SWSF-FP. Later, it was optimized by Chan and Yang [6], and the resulting algorithm was called MFP. They concluded that these algorithms should be used instead of the static Dijkstra algorithm if the percentage of edges changed is smaller than a threshold. FMN [8, 9] and RR [7] are other fully dynamic algorithms. Firigioni et al. [10] compared the performance of these algorithms. They showed that RR has a better performance than FMN due to its efficient cache usage.

Recently, to cope with the complex optimization problems in real-world applications, many algorithms inspired by nature have emerged. Physarum Solver [11] is a popular bio-inspired method to solve the shortest path problem efficiently. This method is inspired by the behavior of a single-celled amoeba-like organism called *Physarum polycephalum*, which is able to find the shortest path in a labyrinth. The mathematical model of *Physarum polycephalum*, which exhibits behavior of path finding in a labyrinth, was described by Tero et al. [12]. Then Miyaji and Ohnishi [13] mathematically proved that Physarum can solve the shortest path problem on Riemann surfaces.

Physarum-related algorithms have various application areas. Liu et al. used the Physarum algorithm to solve the Steiner tree problem [14] in networks. Zhang et al. [15] described a Physarum-related bio-inspired algorithm in transportation networks for identification of critical components. Another Physarum-inspired

algorithm was used to solve the minimal exposure problem in wireless sensor networks in [16]. Zhang et al. [17] combined the Physarum model with the ant colony approach and they called the resulting algorithm PMACO, which is more efficient than the original ant colony algorithm. Zhang et al. [18] proposed another Physarum-related algorithm to solve the 0-1 knapsack problem. Arslan and Manguoglu [19] proposed a parallel Physarum Solver in order to solve the shortest path problem in large graphs. Additionally, different applications of Physarum-related algorithms can be found in [20–22]. When compared to the existing methods, the main advantages of the Physarum algorithm is its adaptivity in network design. Zhang et al. [4] used this property to solve dynamic shortest path problems. In this paper, we reveal another advantage in terms of the solution time using numerical linear algebra and the previous iteration is completely fed into the next one.

3. Background

In this study, dynamic Physarum Solver is proposed to solve the dynamic shortest path efficiently. To present the effectiveness of the proposed method, results are compared with the Δ-stepping algorithm, which is a static implementation of Dijkstra's algorithm. The reason for choosing a static algorithm to compare results is that in our test data, the percentage of changing edges is large, so dynamic algorithms in the literature are infeasible.

3.1. Mathematical model of Physarum

Physarum polycephalum is a single-celled amoeboid organism that is able to find the shortest path connecting food resources in a maze. Tero et al. [11, 12] proposed a mathematical model based on physiological observations of the organism, which is called Physarum Solver. In this model, the network may be considered as a graph. Two special nodes (or vertices), called N_1 and N_n, correspond to the food sources, and other nodes are represented as N_2, N_3, N_4, N_5, and so on. In this representation, N_1 is the source node and N_n is the sink node. The edge connecting N_i to N_j is denoted as M_{ij}. The variable Q_{ij} represents the flux through M_{ij} from N_i to N_j. If it is assumed that the flow through M_{ij} approximates the Poiseuille flow, the flux Q_{ij} is computed by

$$Q_{ij} = \frac{D_{ij}}{L_{ij}}(p_i - p_j), \tag{1}$$

where p_i is the pressure at the node N_i, and D_{ij} and L_{ij} are the conductivity and the length of the edge M_{ij}, respectively. If Kirchhoff's law is applied at each node, Eq. (2) is obtained:

$$\sum_{(i,j)\in M} Q_{ij} = 0, \qquad (j \neq 1, n). \tag{2}$$

The source node N_1 and the sink node N_n hold for the following equations:

$$\sum_i Q_{i1} + I_0 = 0, \tag{3}$$

$$\sum_i Q_{in} - I_0 = 0, \tag{4}$$

where I_0 is the flux from the source node. Next, the graph Poisson equation is represented in Eq. (5) for the pressure:

$$\sum_i \frac{D_{ij}}{L_{ij}}(p_i - p_j) = \begin{cases} I_0 & \text{for } j = 1 \\ -I_0 & \text{for } j = n \\ 0 & otherwise, \end{cases} \tag{5}$$

where $p_n = 0$ (at the basic pressure level), and p_i is determined by solving the system of equations in Eq. (5), which can also be written in matrix form as follows:

$$Ap = b, \tag{6}$$

where p=$(p_1, p_2, ..., p_n)^T$ is the pressure (unknown) and $b = (1, 0, 0, ..., -1)^T$ is the right-hand side vectors, and $A = (A_{ij})$ is a symmetric M-matrix [13] given by Eq. (7):

$$A_{ij} = \begin{cases} \sum_{l \neq i} T_{il} & \text{if } i = j \\ -T_{ij} & otherwise, \end{cases} \tag{7}$$

where $T_{ij} = D_{ij}/L_{ij}$. Now we give the definition of the M-matrix.

Let A be an $n \times n$ square matrix with nonpositive off-diagonal entries. If A can be written as $\alpha I - B$ where B is a nonnegative matrix and $\rho(B) \leq \alpha$, then A is called an M-matrix. M-matrices have some properties such as positivity of principal minors, inverse positivity and splittings, stability, semipositivity, and diagonal dominance [23].

Experimental observations show that while tubes that have larger fluxes are reinforced, those with smaller fluxes are eliminated [12]. Thus, the change of the conductivities is expressed with respect to time as follows:

$$\frac{d}{dt} D_{ij} = f(|Q_{ij}|) - D_{ij}, \tag{8}$$

where $f(|Q_{ij}|)$ is generally given by $f(|Q_{ij}|) = |Q_{ij}|$ as in [12]. To solve Eq. (8), the following equation is obtained by discretizing conductivity values:

$$\frac{D_{ij}^{n+1} - D_{ij}^{n}}{\Delta t} = |Q_{ij}^{n}| - D_{ij}^{n+1}, \tag{9}$$

where Δt indicates the duration of one discrete time step. Becchetti et al. [24] proved that the discretization of the model approximates the shortest path in $\mathcal{O}(|E|L(log|V| + logL)/\epsilon^3)$ iterations where $|V|$ is the number of vertices, $|E|$ is the number of edges, L is the largest length of an edge, and $\epsilon \in (0, 1/300)$.

In the next section, we describe the proposed dynamic algorithm, dynamic Physarum Solver, by focusing on the solution of the linear systems in Eq. (6), which is the most time-consuming step of Physarum Solver.

3.2. Δ-stepping

Δ-stepping, described by Meyer and Sanders [26], is a single source shortest path algorithm. Two classical approaches for solving the single source shortest problem are the Dijkstra and Bellmann–Ford algorithms. When the edge weights of the graph are positive, Dijkstra's algorithm is preferable and runs in $\mathcal{O}(|V|log|V| + |E|)$ time. Dijkstra's algorithm uses a priority queue in which only a single node is processed once per time. On the other hand, Δ-stepping uses a bucket-based priority queue and may be considered as the bucket implementation of Dijkstra's algorithm. The bucket, a one-dimensional array, is a structure of vertices and its size is Δ. Similar to Dijktra's algorithm, a tentative distance (*tentative(v)*) is assigned to each node v in the Δ-stepping algorithm.

Algorithm 1 Δ-stepping Algorithm [25].

Input: A graph given by its vertices V, edges E, cost function L, source vertex s, target vertex t, and Δ parameter
Output: The shortest path from s to t

1: **for** $v_1 \in V$ **do**
2: $tentative(v_1) \leftarrow \infty$
3: $heavy(v_1) \leftarrow \{(v_1, v_2) \in E : L(v_1, v_2) > \Delta\}$
4: $light(v_1) \leftarrow \{(v_1, v_2) \in E : L(v_1, v_2) \leq \Delta\}$
5: **end for**
6: $tentative(s) \leftarrow 0$
7: $k \leftarrow 0$
8: **while** $B \neq \emptyset$ **do**
9: $S \leftarrow \emptyset$
10: **while** $B[k] \neq \emptyset$ **do**
11: $Req \leftarrow \{(v_2, tentative(v_1) + L(v_1, v_2)) : v_1 \in B[k] \wedge (v_1, v_2) \in light(v_1)\}$
12: $S \leftarrow S \cup B[k]$
13: $B[k] \leftarrow \emptyset$
14: RelaxReq(Req)
15: **end while**
16: $Req \leftarrow \{(v_2, tentative(v_1) + L(v_1, v_2)) : v_1 \in S \wedge (v_1, v_2) \in heavy(v_1)\}$
17: RelaxReq(Req)
18: **if** t is processed **then**
19: break
20: **end if**
21: $k \leftarrow k + 1$
22: **end while**

23: **function** RelaxReq(Req)
24: **for** $(u, x) \in Req$ **do**
25: **if** x $<$ tentative(u) **then**
26: B[$tentative(u)/\Delta$] $\leftarrow$ B[$tentative(u)/\Delta$] $\setminus\{u\}$
27: B[x/Δ] $\leftarrow$ B[x/Δ] $\cup\{u\}$
28: $tentative(u) \leftarrow x$
29: **end if**
30: **end for**
31: **end function**

This distance is the weight of a path from the source node to v. The bucket $B[i]$ stores the vertices whose tentative distances are within $[i\Delta, (i+1)\Delta]$. The vertices in the same bucket are processed simultaneously. The pseudocode of the algorithm is given in Algorithm 1. The Δ-stepping algorithm processes each bucket $B[i]$ starting with $B[0]$. In each iteration, all *light* edges (whose weights are less than or equal to Δ) are processed in steps 10–15 in Algorithm 1. This may result in reinsertion in the bucket that is being processed. The *RelaxReq* function is called at the end of each inner *while loop* iteration in step 14 in Algorithm 1 and updates the tentative distance. The inner loop continues until B is empty, and when B is empty, *heavy* edges (whose weights are greater than Δ) are processed in steps 16 and 17 in Algorithm 1. When the shortest path from the source to the target nodes is found, the Δ-stepping algorithm halts. We note that if the single source shortest path is desired, steps 18–20 in Algorithm 1 are ignored. The running time of the algorithm is $\mathcal{O}(log^3|V|/loglog|V|)$ [27].

4. The proposed dynamic algorithm

In this section, we present an iterative algorithm based on Physarum Solver for computing the dynamic shortest path from the source to target vertices, which is called dynamic Physarum Solver. The pseudocode of the algorithm is given in Algorithm 2. In the beginning of the algorithm, D_{ij} is initialized if there is an edge between node i and node j in step 1. Next, the percentage of changing edge (*pce*) parameter is initialized to a number between 0 and 1 and presents how many edges will be updated in the graph. We note that if *pce* equals 0, the original graph is used. In this study, *pce* is set to 0.2. There is a *for loop* between step 4 and step 15 of the algorithm and the graph is dynamically changing in each *for loop* iteration. In each *for loop* iteration, the *pce* is updated by multiplying itself with k in step 5. Thus, the percentage of the changing edges increases with each *for loop* iteration. Then we randomly select pce $\times$ nnz edges where nnz is the total number of nonzero edges in step 6. Next, the edge weights are updated in step 7 and there are different updation mechanisms explained in the following subsection. After the graph is dynamically changed in step 7, Physarum Solver is applied to find the shortest path in the current graph. Physarum Solver includes a *while loop* updating Q_{ij} and D_{ij} values in step 11 and 12, respectively. Moreover, in each *while loop* iteration, it is required to solve the linear systems whose coefficient matrix is a symmetric M-matrix in step 9 and this step is the most time-consuming step of the algorithm. We use the conjugate gradient method as the iterative linear solver and Gauss–Seidel method as the preconditioner for solving such systems, and stopping tolerance is set to 10^{-3} [19].

Iterative methods are preferred for solving large sparse linear systems since they usually require less memory and operation counts than direct methods, especially when an approximate solution of relatively low accuracy is sought. Iterative methods start with an initial guess and try to converge to the exact solution. Therefore, starting with an initial guess close to the exact solution decreases the number of iterations required for solving the linear systems significantly. In our method, the solution vector of the linear systems computed in the previous iteration is used as an initial guess for solving the linear systems in the current iteration. Therefore, a small number of iterations is enough to converge to the exact solution. The complexity of the proposed algorithm is determined by the number of the *for loop* iterations (depending on how many times the

Algorithm 2 Dynamic Physarum Solver

Input: $G = (L,$ source, target) where L is the adjacency matrix of G
Result: The dynamic shortest path from source to target nodes

1: $D_{ij} \leftarrow 1$ where $L_{ij} \neq 0$
2: $pce \leftarrow 0.2$
3: $p \leftarrow 0$
4: **for** k = 0 to n **do**
5: $pce \leftarrow pce \times k$
6: Randomly select **pce** $\times$ **nnz** edges
7: update edge weights (*increase or decrease*)
8: **while** *a termination criterion* **do**
9: **solve** $Ap = b$ iteratively (Eq. (6))
10: $p_n \leftarrow 0$
11: $Q_{ij} \leftarrow \frac{D_{ij}}{L_{ij}}$ x $(p_i - p_j)$
12: $D_{ij} \leftarrow \frac{1}{2}(|Q_{ij}| + D_{ij})$
13: $iter \leftarrow iter + 1$
14: **end while**
15: **end for**

graph changes), the number of the *while loop* iterations (depending on the accuracy of the shortest path), and the cost of solving the linear systems (step 9 in Algorithm 2). When a direct solver is used for solving the linear systems, it takes $\mathcal{O}(|V|^3)$ time. On the other hand, the proposed preconditioned iterative method requires $\mathcal{O}(|E|)$ time per iteration to approach the solution [28]. In the Physarum-related algorithms [11, 12, 14, 29], a direct solver was used to solve such systems and the direct solver is infeasible when the graph size is large. Therefore, the results of these algorithms are shown only using small graphs.

The most important feature of the algorithm is its adaptivity. As the graph dynamically changes, Q and D matrices as well as the p vector computed for finding the shortest path in the previous graph are used in order to compute the shortest path on the current updated graph. Therefore, the proposed method uses information directly coming from the previous computations and it can efficiently find the dynamic shortest path even if the percentage of changing edge weights is large. That is, affected vertices and edges are automatically determined and processed without making any additional computations.

4.1. Obtaining the dynamic graph

In this part, we provide some details about how the graph is dynamically changing in steps 5, 6, and 7 of Algorithm 2. The *pce* presents the percentage of changing edges in the graph. In our implementation, the *for loop* iterates four times, so the graph changes four times, and the *pce* changes between 0 and 0.6. The number of iterations of *for loop* may increase if it is desired that the dynamic graph changes more than four times. In the first *for loop* iteration, *pce* is 0. This means that the original edge weights are used. In the other *for loop* iterations, *pce* is set to 0.2, 0.4, and 0.6 in step 5, respectively. This means that 20, 40, and 60 percent of the edges are randomly chosen in order to change their weights. For instance, if the *pce* is 0.4 and the graph consists of 1000 edges, 400 edges are randomly selected in order to change their weights. In the increasing case, the edge weights are multiplied by 2. For instance, if the edge weight is 100, then the new edge weight will be 200. In the decreasing case, the edge weights are decreased by 20%. For instance, if the edge weight is 100, the new edge weight will be 80 in the decreasing case.

To illustrate how the proposed method works, we give a small example for the increasing case. In this example, critical steps of the proposed method are shown. We point out that the proposed algorithm follows the same steps for the edge weight decreasing case.

Example 1 *An example graph with six vertices (V_1, V_2,...,V_6) and nine edges (Edge1, Edge2,..., Edge9) is given in Figure 1a. The edge weights and labels are shown on the lines. By setting the source vertex as V_1 and the target vertex as V_6, our goal is to find the shortest path from V_1 to V_6 as the graph is dynamically changing. In the beginning of the algorithm, required initialization is performed. In the first for loop iteration, the original graph is used to find the shortest path. Physarum Solver converges to the shortest path by computing Q and D values in the while loop. While the edges whose conductivity values converge to 0 do not form the shortest path, the edges whose conductivity values converge to 1 constitute the shortest path. Thus, the shortest path is V_1-V_2-V_3-V_5-V_6 including Edge1, Edge3, Edge6, and Edge9 as shown in Figure 1b. In the second for loop iteration, the graph dynamically changes and the new graph is shown in Figure 1c. In this graph, two edges, Edge3 and Edge6, are randomly selected and their weights are doubled. Next, Physarum Solver is applied to find the shortest path in the current graph. Our goal is to compute the current shortest path by using previous computations. Since Edge6 and Edge9 are not included in the current shortest path, their conductivity values converge to 0 in the second for loop iteration. We remark that the proposed algorithm automatically*

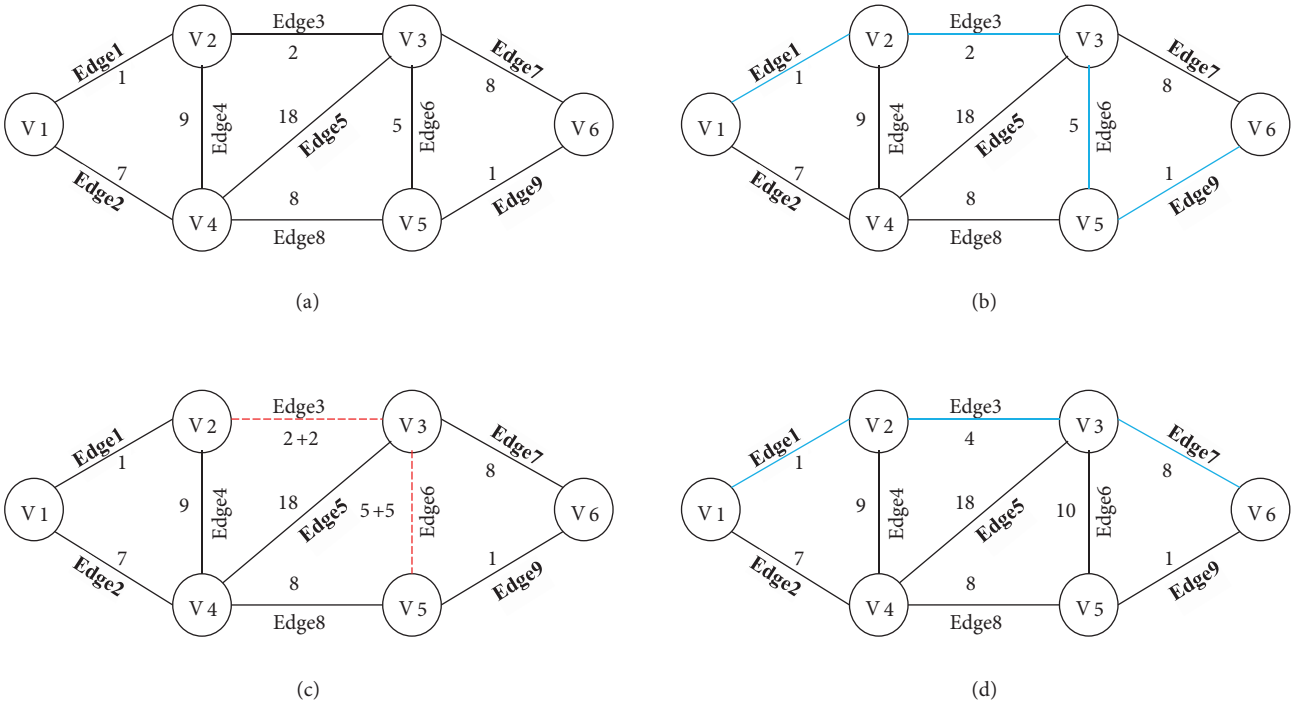

Figure 1. Dynamic Physarum Solver on an example. (a) Simple graph G with six vertices and nine edges. Vertices V_1 and V_6 act as source and target nodes, respectively. (b) The shortest path from source to target nodes is shown by the blue edges. (c) Graph G after modified *Edge3* and *Edge6*. (d) The final shortest path from the source to target nodes is shown by blue lines.

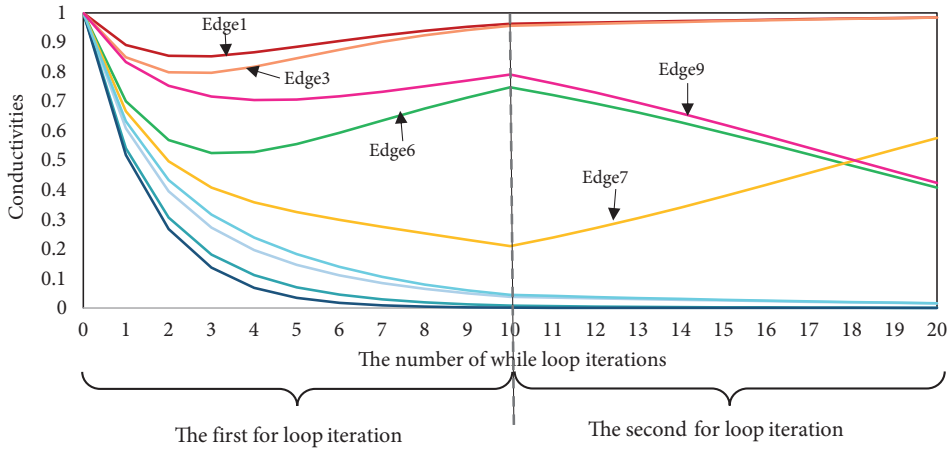

Figure 2. Conductivity values computed by dynamic Physarum Solver on the dynamic graph shown in Figure 1.

determines affected edges and efficiently finds the current shortest path. As a result, the current shortest path is V_1-V_2-V_3-V_6 including Edge1, Edge3, and Edge7 as shown in Figure 1d. The change of the conductivity values (D) for each edge in each while loop iteration is presented in Figure 2. We note that the conductivity values are shown for ten while loop iterations in each for loop iteration. The conductivity values with respect to the second for loop iteration are presented in Figure 2 (where while loop iterations change from 11 to 20). In the other for loop iterations, the dynamic shortest path may be computed in a similar way. In the next section, the results for larger graphs are shown.

5. Experimental evaluation

We present experimental results in terms of time and accuracy and compare them to the Δ-stepping algorithm, which is the baseline algorithm. Δ-stepping is the efficient implementation of the static Dijkstra algorithm. The reason for using the static algorithm for comparison is that when the percentage of changing edges is large, classical dynamic algorithms are known to be infeasible since the processing of affected vertices is too time-consuming [30].

5.1. The dataset

We present experiments with real-world and synthetic graph sets in order to demonstrate the performance of the methods. The properties of the dataset are shown in the Table. While the number of vertices changes from 1M to 2M, the number of edges changes from 2.8M to 100M. Real-world graphs, which are CAL and NW, are obtained from 9th DIMACS Implementation Challenge-Shortest Paths [31]. Synthetic graphs, which are Erdos and Smallworld, are generated by using the Erdos-Renyi and Watts and Strogatz models in the R language, respectively. When comparing the synthetic graphs, the real-world graphs are sparser. The graphs in the dataset have different properties. First, we mention the properties of the real-world graphs. First of all, they are sparser and any two vertices are likely connected with their neighbors. Therefore, degrees of vertices are lower. Moreover, they present power law degree distribution. Second, the Erdos-Renyi random graphs model was presented by Erdos and Renyi [32]. These graphs are produced with respect to two parameters, which are number of vertices and an edge existing probability. They present a binomial degree distribution and degrees of vertices are high. Moreover, they have a low average path length. Third, small-world graphs are described by Watts and Strogatz [33]. The pairs of the vertices in a regular graph are rewired with a probability and the resulting small-world graph has a structure between a regular graph and a random graph. It is possible to connect any two vertices in the graph through just a few edges. This property gives the name 'small-world' to these graphs. They have symmetric degree distribution. Moreover, they are highly clustered, like real-world graphs, and they have low path length, like synthetic graphs. We note that all graphs are undirected, and while we use distances for edge weights in the real-world graphs, we use unit weights in synthetic graphs, namely Smallworld and Erdos.

Table. Graph properties.

Graph name	Vertices	Edges	Model
CAL	~2M	~4.6M	Real-world
NW	~1.2M	~2.8M	Real-world
Erdos	1M	100M	Erdos-Renyi
Smallworld	1M	100M	Small-world

5.2. Computational framework

The numerical results are obtained on a machine that has 2x Intel Xeon E5-2690 V4 2.60 GHz processors and 16 GB memory by the Scientific and Technological Research Council of Turkey. The operating system is CentOS 7.3. The proposed method is implemented in C language and the Portable Extensible Toolkit for Scientific Computation (PETSc) [34] is used for solving the linear systems. PETSc, which is a state-of-the-art library, is developed by the Argonne National Laboratory and includes efficient linear solvers and preconditioners.

The Δ-stepping algorithm is implemented by using the Boost Graph Library [35]. We note that Δ-stepping halts when the shortest path from the source to the target vertices is found without performing any additional computations.

5.3. Experimental results

We evaluate our results based on the solution time and accuracy. Results are investigated for the edge weight increasing and decreasing cases separately. First, we look at results of the edge weight increasing case.

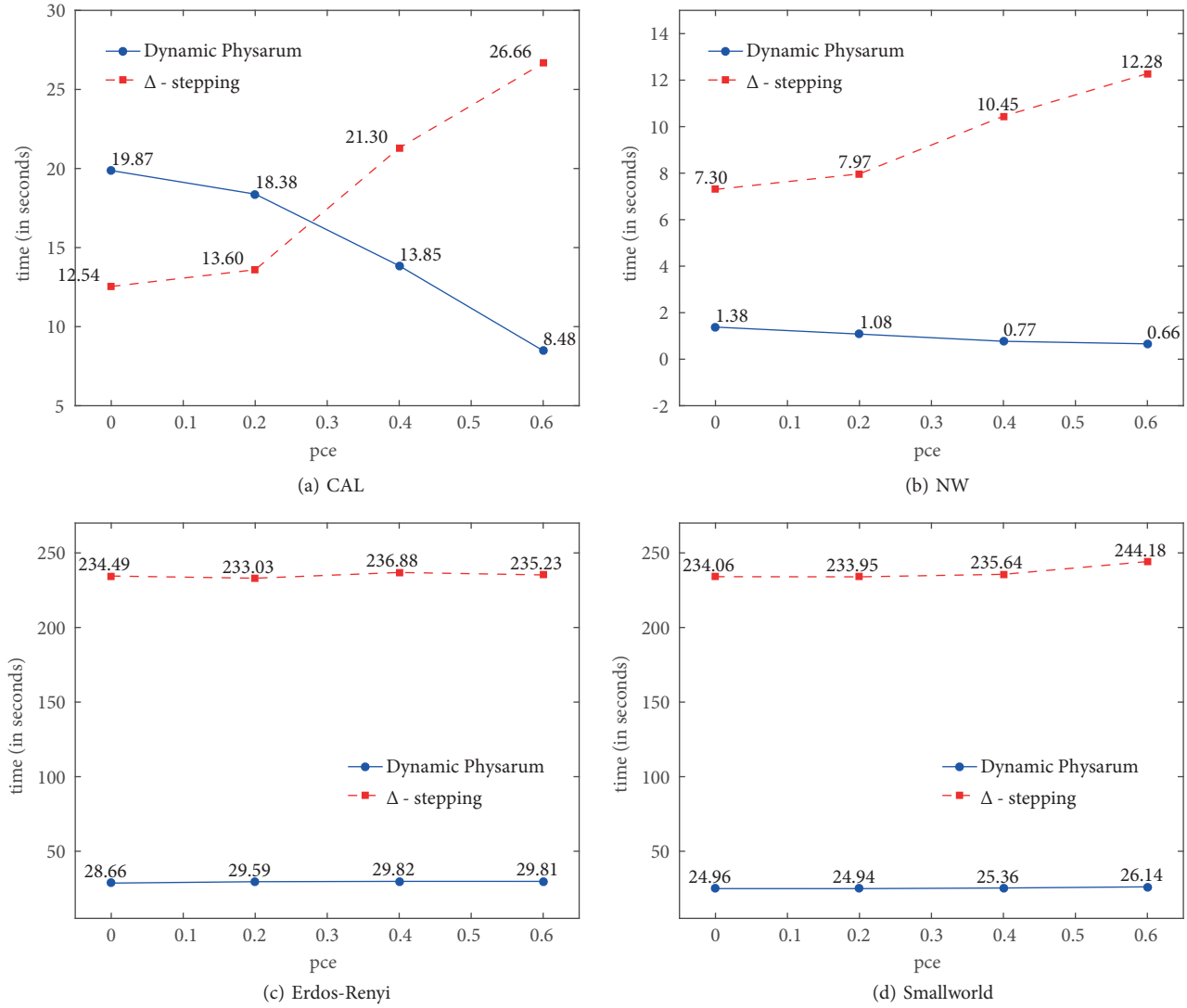

Figure 3. Time results of the increasing case.

Figure 3 presents the solution time results of the dynamic Physarum Solver and Δ-stepping algorithms in seconds for the dataset. First, we look at the results for real-world graphs. Figure 3a presents the solution time results for the CAL graph. As the *pce* is 0, the solution time of Δ-stepping is lower than the solution time of dynamic Physarum Solver. However, when *pce* is 0.6, dynamic Physarum Solver is about 3 times faster than Δ-stepping. Figure 3b presents the solution time results for the NW graph. While the solution time of

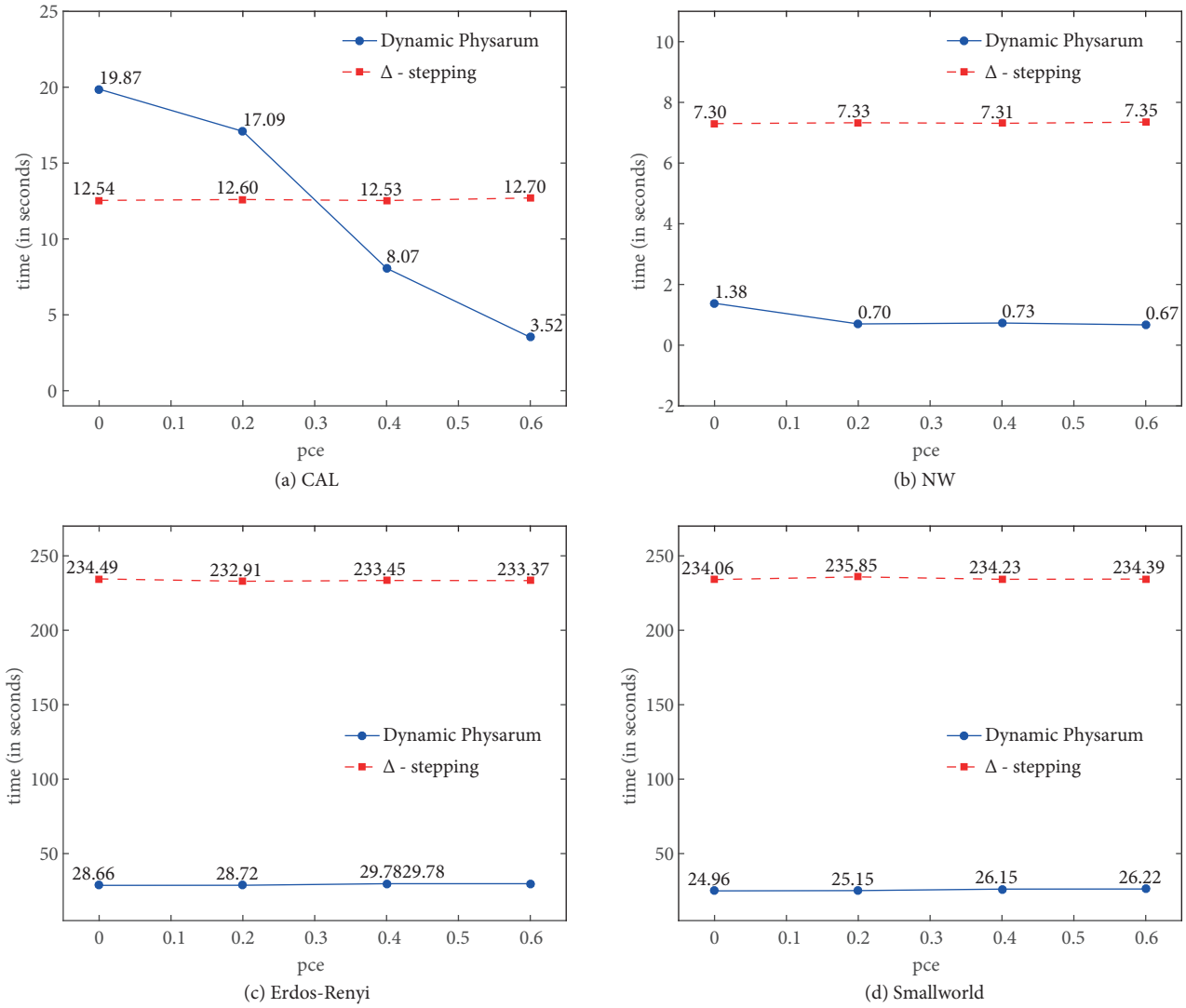

Figure 4. Time results of the decreasing case.

dynamic Physarum Solver is about 5 times faster than Δ-stepping when the *pce* is 0, it is about 18.6 times faster than the baseline when *pce* is 0.6. Moreover, while the solution time of dynamic Physarum Solver decreases, the solution time of Δ-stepping increases for real-world graphs. This presents the efficiency of the proposed method. Figures 3c and 3d present the solution time results for Erdos and Smallworld graphs, respectively. The solution time of dynamic Physarum Solver is 8 and 9 times faster than Δ-stepping for Erdos and Small, respectively. The solution time is about constant for both the dynamic Physarum Solver and Δ-stepping methods.

Now we evaluate the edge weight decreasing case results. Figure 4 presents the solution time for each graph as *pce* increases. For the CAL graph in Figure 4a, while the solution time of Δ-stepping is lower than the solution time of dynamic Physarum Solver when *pce* is 0, it is about 3.6 times higher than the solution time of dynamic Physarum Solver when *pce* is 0.6. Moreover, the solution time of dynamic Physarum Solver decreases as *pce* increases, similar to the increasing case. For the NW graph in Figure 4b, the proposed method is about 10 times faster than Δ-stepping on average. We note that as the *pce* increases, the solution time of Δ-stepping is almost constant for real-world graphs while it increases for the edge weight increasing case. For

synthetic graphs, the solution time of both methods is almost constant, and the proposed method is 7.5 and 9 times faster than Δ-stepping as shown in Figures 4c and 4d for Erdos and Smallworld graphs, respectively.

In summary, as the graph is dynamically changing, while the solution time of the proposed method decreases for CAL and NW real-world graphs, the solution time of Δ-stepping increases. This is due to the fact that dynamic Physarum Solver uses the information coming from previous iterations directly. Furthermore, in the solution of the linear systems, using the previous solution vector as an initial guess to solve the linear systems in the current iteration decreases the solution time. The number of iterations for solving the linear systems in step 9 of Algorithm 2 is shown in Figure 5 for CAL and NW graphs for increasing and decreasing cases separately. When *pce* is 0.6, a small number of iterations is enough to converge the desired solution for the CAL and NW graphs. Moreover, decreasing the edge weights has a positive effect on the solution time for the proposed method. As shown in Figure 5, the edge weight decreasing case requires lower iteration counts when compared to the increasing case for CAL and NW graphs. The reason for this is that in real-world graphs, there are many edges in sparse regions with large weights. If the edges with large weights are randomly selected to increase their weights, then large weights become larger. As the graph is dynamically changing, they are largely unbalanced. However, in the decreasing case, the graph approaches the uniform distribution with decreasing large weights as the graph is dynamically changing. Therefore, solving the linear systems in the increasing case is more difficult than in the decreasing case. On the other hand, the reason for increasing the solution time of the Δ-stepping method in the edge weight increasing case is that real-world graphs are largely unbalanced as the graph is dynamically changing. This causes many reinsertions in the priority queue and increases the solution time. Similar to the proposed method, decreasing of the edge weights has a positive effect when compared to the increasing case for the Δ-stepping method. We recall that in the edge weight decreasing case, the solution time of Δ-stepping is almost constant for real-world graphs.

For synthetic graphs, the solution time of dynamic Physarum Solver is almost constant. The main reason for this is that solving the linear systems requires about the same amount of iterations and $\sim$20 iterations are enough to converge to the desired solution. Similarly, the solution time of Δ-stepping is almost constant since synthetic graphs have uniform edge weight distribution as the graph is dynamically changing.

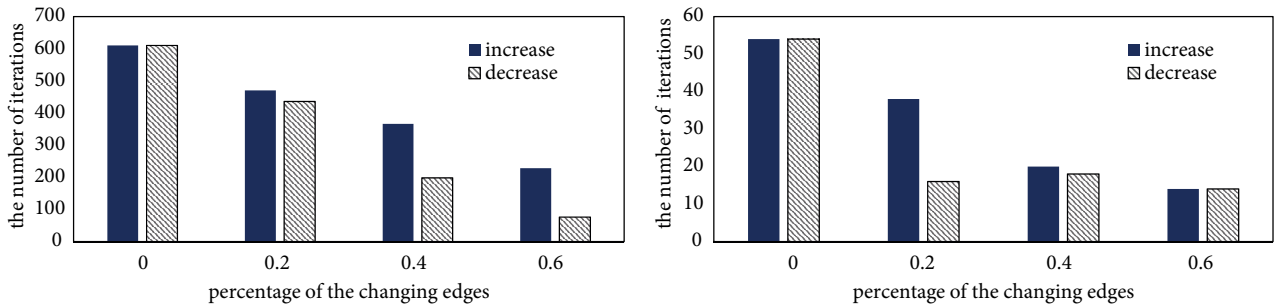

Figure 5. The number of iterations to solve the linear systems shown in step 9 in Algorithm 2.

Next, we look into the results in terms of accuracy. The accuracy of dynamic Physarum Solver depends on both the number of *while loop* iterations and the accuracy of the solution of linear systems. The number of *while loop* iterations is set to three and stopping tolerance, which is the residual norm relative to the norm of the right-hand side, is set to 10^{-3}. Figure 6 presents computed dynamic shortest path lengths by dynamic Physarum Solver and Δ-stepping for increasing and decreasing cases separately. Δ-stepping and dynamic Physarum Solver compute nearly the same shortest path lengths.

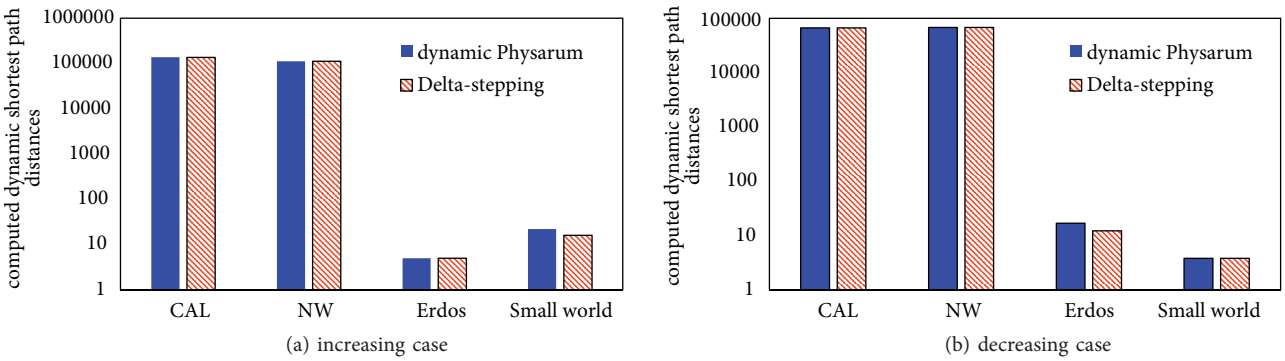

Figure 6. Computed shortest path distances for both Δ-stepping and Dynamic Physarum Solver.

6. Conclusion

Dynamic Physarum Solver is presented for computing the shortest path in dynamically changing graphs efficiently. Experimental results are performed on large synthetic and real-world graphs and compared to a state-of-the-art implementation of Dijkstra's algorithm, Δ-stepping in the Boost Graph Library. Results show that the proposed method efficiently computes the dynamic shortest path even if the number of changing edges is large. An important reason for the better performance of dynamic Physarum Solver is that as the graph dynamically changes, it distinguishes dynamically changing edges and affected vertices and recomputes spontaneously. Another important reason for the better performance of dynamic Physarum Solver is that array-based data structures are more suitable for the multiple levels of cache hierarchy as well as using an initial guess and an effective linear solver with the preconditioner for solving the linear systems.

References

[1] King V. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In: 40th Annual Symposium on Foundations of Computer Science; Washington, DC, USA; 1999. p. 81–89.

[2] Demetrescu C, Italiano GF. Fully dynamic all pairs shortest paths with real edge weights. In: Proceedings 2001 IEEE International Conference on Cluster Computing; Las Vegas, Nevada, USA; 2001. p. 260–267.

[3] Narvaez P, Siu KY, Tzeng HY. New dynamic algorithms for shortest path tree computation. IEEE/ACM Transactions on Networking 2000; 8 (6): 734–746.

[4] Zhang X, Chan FTS, Yang H, Deng Y. An adaptive amoeba algorithm for shortest path tree computation in dynamic graphs. Information Sciences 2017; 405: 123–140.

[5] Narvaez P, Siu KY, Tzeng HY. New dynamic SPT algorithm based on a ball-and-string model. IEEE/ACM Transactions on Networking 2001; 9 (6): 706–718.

[6] Chan EPF, Yang Y. Shortest Path Tree Computation in Dynamic Graphs. IEEE Transactions on Computers 2009; 58 (4): 541–557.

[7] Ramalingam G, Reps T. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. Journal of Algorithms 1996; 21 (2): 267–305.

[8] Frigioni D, Marchetti-Spaccamela A, Nanni U. Fully Dynamic Algorithms for Maintaining Shortest Paths Trees. Journal of Algorithms 2000; 34 (2): 251–281.

[9] Frigioni D, Marchetti-Spaccamela A, Nanni U. Fully Dynamic Output Bounded Single Source Shortest Path Problem. In: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms; Atlanta, Georgia, USA; 1996. p. 212–221.

[10] Frigioni D, Ioffreda M, Nanni U, Pasqualone G. Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Paths Problem. ACM Journal on Experimental Algorithmics 1998; 3 (5).

[11] Tero A, Kobayashi R, Nakagaki T. Physarum solver: A biologically inspired method of road-network navigation. Physica A: Statistical Mechanics and its Applications 2006; 363 (1): 115–119.

[12] Tero A, Kobayashi R, Nakagaki T. A mathematical model for adaptive transport network in path finding by true slime mold. Journal of Theoretical Biology 2007; 244 (4): 553–564.

[13] Miyaji T, Ohnishi I. Physarum Can Solve the Shortest Path Problem on Riemann Surface Mathematically Rigorously. International Journal of Pure and Applied Mathematics 2008; 47 (3): 353–369.

[14] Liu L, Song Y, Zhang H, Ma H, Vasilakos AV. Physarum Optimization: A Biology-Inspired Algorithm for the Steiner Tree Problem in Networks. IEEE Transactions on Computers 2015; 64 (3): 818–831.

[15] Zhang X, Adamatzky A, Yang H, Mahadaven S, Yang XS, Wang Q, et al. A bio-inspired algorithm for identification of critical components in the transportation networks. Applied Mathematics and Computation 2014; 248: 18–27.

[16] Liu L, Song Y, Ma H, Zhang X. Physarum optimization: A biology-inspired algorithm for minimal exposure path problem in wireless sensor networks. In: 2012 Proceedings IEEE INFOCOM; Orlando, FL, USA; 2012. p. 1296–1304.

[17] Zhang Z, Gao C, Liu Y, Qian T. A universal optimization strategy for ant colony optimization algorithms based on the Physarum -inspired mathematical model. Bioinspiration and Biomimetics 2014; 9 (3):036006.

[18] Zhang X, Huang S, Hu Y, Zhang Y, Mahadevan S, Deng Y. Solving 0-1 knapsack problems based on amoeboid organism algorithm. Applied Mathematics and Computation 2013; 219 (19): 9959–9970.

[19] Arslan H, Manguoglu M. A parallel bio-inspired shortest path algorithm. Computing 2018; doi: 10.1007/s00607-018-0621-x.

[20] Zhang X, Zhang Z, Zhang Y, Wei D, Deng Y. Route selection for emergency logistics management: A bio-inspired algorithm. Safety Science 2013; 54 :87–91. doi: 10.1016/j.ssci.2012.12.003.

[21] Zhang X, Adamatzky A, Chan FTS, Deng Y, Yang H, Yang XS, et al. A Biologically Inspired Network Design Model. Scientific Reports 2015; 5 (10794). doi: 10.1038/srep10794.

[22] Zhang X, Zhang Y, Deng Y. An improved bio-inspired algorithm for the directed shortest path problem. Bioinspiration & Biomimetics 2014; 9 (4):046016.

[23] Plemmons RJ. M-matrix characterizations.I-nonsingular M-matrices. Linear Algebra and its Applications 1977; 18 (2):175–188.

[24] Becchetti L, Bonifaci V, Dirnberger M, Karrenbauer A, Mehlhorn K. Physarum Can Compute Shortest Paths: Convergence Proofs and Complexity Bounds. In: ICALP'13 Proceedings of the 40th international conference on Automata, Languages, and Programming - Volume Part II; Riga, Latvia; 2013. p. 472–483.

[25] Madduri K, Bader DA, Berry JW, Crobak JR. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-scale Graph Instances. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments; Philadelphia, PA, USA; 2007. p. 23–35.

[26] Meyer U, Sanders P. Δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms 2003; 49:114–152.

[27] Meyer U, Sanders P. Δ-stepping: A Parallel Single Source Shortest Path Algorithm. In: European Symposium on Algorithms; Venice, Italy; 1998. p. 393–404.

[28] Shewchuk JR. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Pittsburgh, PA, USA: Technical Report, School of Computer Science, Carnegie Mellon University; 1994.

[29] Wang Q, Lu X, Zhang X, Deng Y, Xiao C. An anticipation mechanism for the shortest path problem based on Physarum polycephalum. International Journal of General Systems 2015; 44 (3): 326–340.

[30] Liu X, Wang H. Dynamic Graph Shortest Path Algorithm. In: Web-Age Information Management; Harbin, China; 2012. p. 296–307.

[31] Demetrescu C, Goldberg AV, Johnson DS. Implementation Challenge for Shortest Paths. Boston, MA: Springer US; 2008.

[32] Erdos P, Renyi A. On the evaluation of random graphs. Mathematical Institute of the Hungarian Academy of Sciences 1960; 5:17–61.

[33] Watts DJ, Strogatz SH. Collective dynamics of small-world networks. Nature 1998; 393 (6684):440–442.

[34] Balay S, Gropp WD, McInnes LC, Smith BF. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. Cambridge, MA, USA: Birkhäuser Press; 1997.

[35] Edmonds N, Breuer A, Gregor D, Lumsdaine A. Single-Source Shortest Paths with the Parallel Boost Graph Library. In: The Ninth DIMACS Implementation Challenge - The Shortest Path Problem; Piscataway, NJ; 2006. p. 219–248.