Bulgarian Chemical Communications, Volume 50, Special Issue B, (pp. 115 – 120) 2018

Optimization algorithms for finding the shortest paths

F. I. Sapundzhi*, M. S. Popstoilov

South-West University Neofit Rilski, 2700 Blagoevgrad, Bulgaria

Received: July 15, 2017; Revised: October 17, 2017

Graph traveling problems are among the oldest problems of graph theory. The shortest path algorithms are intensively studied problems, which have a lot of applications such as: many problems of dynamic programming with discrete state and discrete time; network optimization problem-networks of roads and telecommunication networks, etc. At the present time the graphs provide simple but often useful formal representation of biological networks capturing one-to-one relationships between biological units. The aim of the present work is to evaluate the Dijkstra's algorithm, Floyd-Warshall algorithm, Bellman-Ford algorithm, and Dantzig's algorithm in solving the shortest path problem, that can be applied to very different biological systems and problems for biological system modelling. A brief overview of the different types of algorithms for finding the shortest paths is given. C# implementation of the considered algorithms are presented to show how works each of them. The results of evaluating the algorithms along with their time complexity are shown.

Keywords: Molecular modelling, Shortest path problem, Dijkstra's algorithm, Floyd-Warshall algorithm, Bellman-Ford algorithm, Dantzig's algorithm.

INTRODUCTION

With the advent of computational biology, biological information is more often represented and stored in the form of biological interactions between genes, proteins, miRNAs, etc. The biological data assay has developed from understanding the function of single genes to interpreting the collective behaviour of complex biological systems, which can be modelled and analysed in the form of protein-ligand interaction networks, gene regulatory networks and others. In the fields of biology and medicine applications of network analysis include identification of drug target, determining the functions of proteins and genes, designing effective strategies for treating various diseases, providing early diagnosis of disorders, etc [1].

Networks of protein-protein interaction, biochemical networks, transcriptional regulation networks, signal transduction are the highlighted network categories in systems biology often sharing characteristics [2].

Signal transduction networks often use graphs to represent a series of interactions between proteins, chemicals or macromolecules. Databases that store information about signal transduction pathways are MiST [3], TRANSPATH [4], etc.

To operate the plenty of information, complex computational methods are needed to manage the number and size of biological networks which can be represented in the form of mathematical graphs.

Many transport, distribution tasks, tasks for selecting optimal routes or situation of service centers, tasks for making schedules, are described by the language of graphs and networks. Series of physical, chemical, economical and managing systems are successfully interpreted and examined using graph theory.

In many cases tasks are linear – target function is linear, as well as all of the constraints, which means that they can be solved using linear optimization. The real tasks (these who fully enough affect the reality) are too complex, which causes searching efficient and flexible algorithms regarding output data. Graph theory gives good opportunities for this [1,2,5,6].

The objectives of this research paper are: (a) to determine and identify the concepts of the shortest path problem; (b) to determine the representation of graphs in computer in order to solve the shortest path problem, as well as to explore and understand the different basic terms of graphs; (c) to explain the general concepts and the C# implementations of Dijkstra's algorithm, Floyd-Warshall algorithm, Bellman-Ford algorithm and Danzig's algorithm; (d) to evaluate each algorithm, and to present the evaluations' results.

MATERIALS AND METHODS

Graph theory and definitions

The shortest path problem is a task for finding the shortest path or route from a starting point to a final destination. In order to represent the shortest path problem we use graph theory. To introduce the basic concepts of it, we give the empirical and the mathematical description of graphs that represent networks as they are originally defined in the literature [5,6].

* To whom all correspondence should be sent.
E-mail: sapundzhi@swu.bg

© 2018 Bulgarian Academy of Sciences, Union of Chemists in Bulgaria

A *graph* G is a pair (V, E), where V is a finite set of vertices and E is a set of connections (edges) between the vertices. An edge $e = (u, v)$ consists of two vertices such that $u, v \in V$.

A graph $G = (V, E)$ is *edge-weighted,* if each edge $e \in E$ *has a weight,* $W(e) \in \mathbb{R}$. Let $G = (V, E)$ denotes an edge-weighted graph with real edge weights $W(e), e \in E$. We will say that D is a *metric* for G if, for any three vertices $u, v, w \in V$, $D(u, v) = D(v, u) \geq 0$, with $D(u, v) = 0$ if and only if $u = v$, and $D(u, v) \leq D(u, w) + D(w, u)$. One way of defining metric distance on a weighted graph is to use the *shortest-path* metric $\delta(\ldots)$ on the graph or its sub graph, i.e., $(D(u, v) = \delta(u, v)$, the shortest path distance between u and v for all $u, v \in V$. We will say that the edge weighted tree $T = T_G(V', E')$ is a *tree metric* for G, with respect to distance function D, if for any pair of vertices u, v in G, the length of the unique path between them in T is equal to $D(u, v)$. An *ultra-metric* is a special type of tree metric defined on rooted trees, where the distance to the root is the same for all leaves in the tree, an approximation that introduces small distortion. A metric D is an ultra-metric if, for all pointsx, y, z we have $D[x, y] \leq \max\{D[x, z], D[y, z]\}$. An ultra-metric does not satisfy all the properties of a tree metric distance. To create a general tree metric from an ultra-metric, we need to satisfy the *4-point* condition:

$$D[x, y] + D[z, w] \leq \max\{D[x, z] + D[y, w], D[x, w] + D[y, z]\}, \text{ for all } x, y, z, w.$$

A metric that satisfies the 4-point condition is called an *additive metric*.

If routes are one-way then the graph will be directed or else it will be undirected. In the literature are presented many different types of algorithms that solve the shortest path problem. Only several of the most popular conventional shortest path algorithms are going to be discussed in this paper, and they are as follows: Dijkstra's algorithm, Floyd-Warshall algorithm, Bellman-Ford algorithm and Dantzig's algorithm [8-11].

EXPLANATION AND IMPLEMENTATION OF THE ALGORITHMS

Explanation and implementation of Dijkstra's algorithm

Explanation:

Step 1: Mark the initial vertex (s). Let: $d(s) = 0$ (Constant distance label) $d(x) = \infty$ (Tentative distance label) $p = s$ (p – last marked vertex).

Step 2: (Changing the tentative distance labels). For all unmarked vertices x the numbers $d(x)$ are recalculated by Eqn.1:

$$d(x) = \min\{d(x), d(p) + c(p, x)\} \qquad (1)$$

where $d(x)$ is the minimal tentative distance from s to x, $d(p)$ – minimal tentative distance from s to p and $c(p, x)$ is the edge's weight from p to x. (Obviously we may change only those $d(x)$, for which the edge (p, x) exists, the rest of the numbers remain the same).

If $d(x) = \infty$, for each unmarked vertex x, stop the procedure – it means that there are no paths from s to all unmarked vertices. Otherwise mark the one vertex x which has minimal distance label $d(x)$. Also color the edge which goes into vertex x, for which the minimal distance label from Eqn.1 is reached. Let $p = x$.

Step 3: If $p = t$, the procedure ends, the only way from s to t, made out of marked edges, is the shortest path between s and t. Otherwise, go back to Step 2 [9].

Implementation:

The following basic variables are used: $Dictionary < V, double > distances$ -stores the distances from source vertex to all vertices; $Dictionary < V, E > bestVertexEdg$ -stores an edge for every vertex that minimizes its distance label; $Dictionary < V, bool > marked$ -if some vertex is marked, its value in this dictionary is true; $List < E > markedEdges$ -list of all marked edges; $V\ currentVertex$ - last marked vertex [7].

The algorithm of Dijkstra determines the shortest path and its length from a given vertex s to a target vertex t. It is supposed that all edge's lengths are positive. The algorithm stops: (1) if there's no path from s to t; (2) if one or more edges have negative weights; (3) when the target vertex t is marked.

Step 1: A tentative distance for all vertices in the given graph that represents the minimal distance from the source vertex to all vertices and use $Dictionary < V, double > distances$ for these labels. Initially, all labels are set to big enough number to represent infinity but s ($distances[s] = 0$) and ($distances[v] = inf, v \neq s$) which means that the path lengths from s to the rest vertices are unknown, after that mark source vertex s as visited and set it as current vertex ($currentVertex := s$).

Step 2: The distance labels ($distances[v]$) for each unmarked neighbor vertex v where an edge exists from $currentVertex$ to v are recalculated by Eqn. 2:

$$distances[v] = \min\{distances[v], distances[currentVertex] + c(currentVertex, v)\} \qquad (2)$$

where c $(currentVertex, v)$ is edge's weight from $currentVertex$ to v. Then set $currentVertex$ this

unmarked vertex v which has minimal distance label and color the edge entering v for which the minimal number from Eqn. 2 is reached.

Step 3: If current vertex is target ($currentVertex = t$) the procedure ends. The only path from s to t, made out of marked edges, is the shortest path between s and t. Otherwise, go back to Step 2.

Dijkstra's algorithm using adjacency matrix: In this case following variables are used: $double[,]\ adjacencyMatrix$ – graph's adjacency matrix; $Dictionary < V, int >\ verticesIndices$ - stores the corresponding index of every vertex in the adjacency matrix; $int\ verticesCount$-number of all vertices; $int\ sourceIndex$- starting vertex index; $int\ targetIndex$ - destination vertex index; $int\ currentVertexInde$-last marked vertex index; $double[]\ distances$ – stores the distances from source vertex to all vertices. All vertices are indexed with the numbers from 0 to $verticesCount - 1$; $List < E >\ bestVertexEdg$- stores an edge for every vertex that minimizes its distance label; $bool\ []\ marked$ – if $marked[i]$ is true then the vertex with index i is marked; $List < E >\ markedEdges$ - list of all marked edges [7].

Step 1: Convert the graph to adjacency matrix (a square matrix) with dimensions $n\ x\ n$ (n – number of vertices). The matrix elements are calculated by Eqn. 3:

$$A_{ij} = \begin{cases} \infty\ if\ there\ isn't\ an\ edge\ from\ i\ to\ j \\ minimal\ edge's\ weight\ from\ i\ to\ j \end{cases} \quad (3)$$

where $A_{ii} = 0, \forall i$. A hash table which keeps the index of every vertex in the adjacency matrix is created, that so the path could be restored. A tentative distance label which represents the minimal current distance from vertex s to the rest vertices v is assigned for each vertex in the graph. For this purpose, an array $distances[]$ is used. The array is initialized as follows: $distances[s] = 0$, $distances[v] =$ big enough number where $v \neq s$ (s and v – array's indices). The vertex s was marked as visited and set it as current vertex using its index.

Step 2: The distance labels ($distances[v]$) are recalculated for each unmarked neighbor vertex v where an edge exists from $currentVertex$ to v by Eqn. 4:

$$distances[v] = \min \left\{ \begin{matrix} distances[vIndex], \\ distances[currentVertexIndex] \\ +c(currentVertexIndex, v) \end{matrix} \right\} \quad (4)$$

where $vIndex$ and $currentVertexIndex$ are respectively the distances array's indices of vertices v and current marked vertex. Then it is appropriated $currentVertexIndex$ the index of this unmarked vertex v which has minimal distance label and colored the edge entering v for which the minimal number from formula were reached (Eqn. 4).

Step 3: If the current vertex index equals the target vertex index ($currentVertexIndex = targetVertexIndex$) the procedure ends. The only path from s to t, made out of marked edges, is the shortest path between s and t. Otherwise go back to Step 2.

Explanation and implementation of Ford's algorithm

Explanation:

The Bellman-Ford algorithm is Dijkstra's algorithm modification in case that some edges have negative weights [9]. The Bellman-Ford algorithm's modification consists of: in Step 2 the distance labels $d(x)$ for all vertices are recalculated. If the distance label $d(x)$ of some vertex x can be changed, the distance is updated to the new lower value and if this vertex x is marked, its marking and the incident with it colored edges are ignored. The algorithm stops when all vertices are marked and after *S*tep 2 none of the distance labels $d(x)$ has changed. This algorithm is slower than Dijkstra's. As it admits edges with negative weights, a graph can contain negative length cycle. In case like this, the algorithm won't work properly.

Implementation:

Instead of $Dictionary < V, bool >\ marke$, here $HashSet < V >\ markedVertices$ (set of all marked vertices) and $HashSet < V >\ unmarkedVertices$ (set of unmarked vertices) are used. The $distances[v]$ for every vertex in the graph (unlike Dijkstra's algorithm the labels of unmarked vertices are recalculated) are recalculated.

Explanation and implementation of Floyd's algorithm

Explanation:

This algorithm finds the shortest path length between every couple of vertices. Edges can have negative weights but loops with negative length are not allowed [10].

Step 1: All vertices are numbered with the numbers from 1 to n. Matrix $D^0 = (d_{ij}^0)_{nxn}$ is determined. Element (i, j) is the shortest edge's length (with least weight) between i and j. $d_{ij}^0 = \infty$ if (i, j) edge is missing and $d_{ii}^0 = 0, \forall i$.

Step 2: For each $m \in [1, n]$ are determined the matrix elements $D^m = (d_{ij}^m)_{nxn}$ by the matrix elements $D^{m-1} = (d_{ij}^{m-1})_{nxn}$ using Eqn. 5:

$$d_{ij}^m = min\{d_{ij}^{m-1}, d_{im}^{m-1} + d_{mj}^{m-1}\} \quad (5)$$

Every element (i, j) in the matrix D^n is the shortest path's length from i to j.

Implementation:

Step 1: Convert the graph to adjacency matrix (a square matrix) with dimensions n x n (n – number of vertices). The matrix' elements are calculated by Eqn. 6:

$$d_{ij}^0 = \begin{cases} \infty \; if\ there\ isn't\ an\ edge\ from\ i\ to\ j \\ minimal\ edge's\ weight\ from\ i\ to\ j \end{cases} \quad (6)$$

where $d_{ii}^0 = 0, \forall i$. A hash table which keeps the index of every vertex in the adjacency matrix is created, so the path could be restored [7].

Step 2: Three nested for-loops are used to represent Eqn.5:

$$Dm[i,j] = Math.Min(prevD[i,j], prevD[i, m-1] + prevD[m-1, j])$$

Explanation and implementation of Danzig's algorithm

Explanation:

Step 1: All vertices are numbered with the numbers from 1 to n. Matrix $D^0 = (d_{ij}^0)_{nxn}$ is determined. The element (i, j) is the shortest edge's length (with least weight) between i and j [11]. The elements $d_{ij}^0 = \infty$ if (i, j) edge is missing and $d_{ii}^0 = 0$ for every i.

Step 2: The matrix D^m for each $m = 1, 2, \dots, n$ using D^{m-1} and D^0 are determined by the following equations:

$$d_{ii}^m = 0 \; for\ each\ i\ and\ each\ m \quad (7)$$

$$d_{ij}^m = \min\{d_{ij}^{m-1}, d_{im}^m + d_{mj}^m\}, when\ i,j = 1,2,\dots,m-1 \quad (8)$$

$$d_{im}^m = \min_{j=1,2,\dots,m-1}\{d_{ij}^{m-1} + d_{jm}^0\}, when\ i = 1,2,\dots,m-1 \quad (9)$$

$$d_{mj}^m = \min_{i=1,2,\dots,m-1}\{d_{mi}^0 + d_{ij}^{m-1}\}, when\ j = 1,2,\dots,m-1 \quad (10)$$

This algorithm performs the same operations as Floyd's algorithm but in other order. In this case the matrix D^m $(m \geq 1)$ has dimensions $m \times m$.

Implementation:

For this algorithm following variables are used: $double[,]\ adjacencyMatrix$ -stores the adjacency matrix which elements are described below; $Dictionary < V, int >\ verticesIndices$ - stores the corresponding index of every vertex in the adjacency matrix; $int\ verticesCount$ - number of all vertices in the graph; $double[,]\ D0$ - matrix D^0; $double[,]\ Dm$ -current matrix $D^m\ (m = 1 \dots n)$; $double[,]\ prevDm$ -represents matrix D^{m-1}.

Step 1: Convert the graph to adjacency matrix (a square matrix) with dimensions $n \times n$ (n -number of vertices) [7]. The matrix elements are

$$d^0{}_{ij} = \begin{cases} \infty,\ if\ there\ isn't\ an\ edge\ from\ i\ to\ j \\ minimal\ edge's\ weight\ from\ i\ to\ j \end{cases} \quad (11)$$

$d^0{}_{ii} = 0, \forall i$. We also create a hash table which keeps the index of every vertex in the adjacency matrix so we can restore the path.

Step 2: The matrix D^m for each $m = 1 \dots n$ is determined by the equations for $Dm[i, m-1]$ and $Dm[m-1, j]$ $(i, j = 0 \dots m-2)$:

$$Dm[i, m-1] = Math.Min(Dm[i, m-1], prevDm[i,j] + D0[j, m-1]) \quad (12)$$

$$Dm[m-1, j] = Math.Min \quad (13)$$

The elements of $Dm[i,j]$ $(i, j = 0 \dots m-2)$ depend on the upper ones $Dm[i,j] = Math.Min(prevDm[i,j], Dm[i, m-1] + Dm[m-1, j])$.

Table 1. The values of execution time and different number of vertices for shortest path algorithms (Dijkstra, Ford, Floyd, Dantzig).

№	Vertices	Edges	Dijkstra's algorithm	Dijkstra's algorithm by adjacency matrix	Ford's algorithm	Floyd's algorithm	Dantzig's algorithm
1	100	10 000	0.142	0.066	0.042	0.341	0.387
2	500	50 000	1.165	0.504	0.584	32.123	35.229
3	500	100 000	1.036	0.769	0.755	34.307	36.981
4	500	1 000 000	3.03	5.209	2.707	37.406	45.805
5	1 000	100 000	1.468	0.64	3.238	266.92	324.216
6	2 500	1 000 000	15.764	9.753	14.622	993.166	1985.166
7	5 000	500 000	24.813	5.958	61.076	-	-
8	5 000	1 000 000	48.685	4.987	17.488	~13620	-
9	5 000	1 000 000	35.545	12.706	69.894	-	-
10	10 000	100 000	68.81	19.942	75.975	-	-
11	10 000	1 000 000	84.742	17.783	80.715	-	-
12	25 000	5 000 000	506.299	-	-	-	-
13	25 000	5 000 000	2259	-	-	-	-
14	25 000	5 000 000	-	-	757.885	-	-
15	25 000	5 000 000	-	-	1232.543	-	-

RESULTS AND DISCUSSION

The algorithms of Dijkstra [8], Floyd-Warshall [9], Bellman-Ford [10] and Dantzig [11] for finding the shortest path were tested using Visual Studio Community 2015, Intel® Pentium® Processor N3710, 1.6 GHz (4 CPUs), 4096 MB RAM [7,13]. The values of execution time and different number of vertices for shortest path algorithms (Dijkstra, Ford, Floyd, Dantzig) are presented in Table 1The experimental results are shown in Table 2. Time complexity of the shortest path algorithms depends on the number of vertices, number of edges and edge length. As can be seen, the time complexity of the Dijkstra's algorithm depends on the number of vertices and is inversely proportional to the number of vertices. The time complexity was higher for Bellman-Ford algorithm than Dijkstra's algorithm. For higher number of nodes, the Dijkstra's algorithm is better and efficient (Table 3) [14-17].

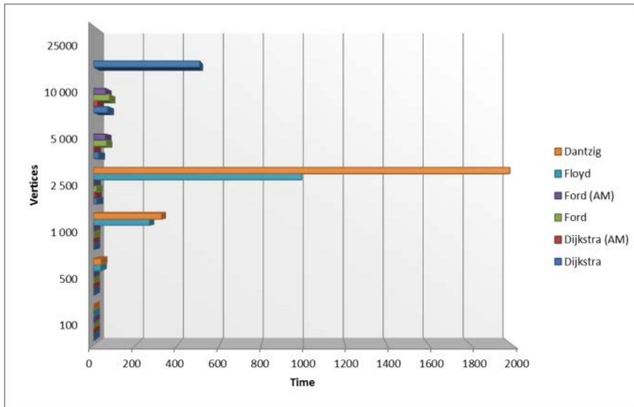

Fig. 1. Execution time and different number of vertices for the shortest path algorithms - Dijkstra, Ford, Floyd and Dantzig.

As it can be seen from the results in Table 2, the Dijkstra's algorithm and its implementation for adjacency matrix in C # representation of graphs provide better performance in the cost of memory. The time complexity for the matrix representation is O(V^2).

The algorithms for finding the shortest paths – Dijkstra's, Ford's, Floyd's and Dantzig's were examined and analyzed. The best results for the values of execution time and different number of vertices are obtained by Dijkstra's algorithm (Table 2 and Figure 2). This algorithm is also implemented through a adjacency matrix. As can be seen in Figure 2, better results are obtained when using an adjacency matrix for the same input parameters.

A C# implementation for drawing the shortest path for Dijkstra's algorithm was developed. The software draws and marks the nodes and edges of the finding shortest path by coloring them in red. The algorithm of Dijkstra is implemented and visually demonstrated in Visual Studio Community 2015 [7,13]. Figure 3 shows C # implementation of some examples of Dijkstra's algorithm. Graphic representation is going to be implemented for the rest algorithms for finding the shortest paths.

Table 3. Dijkstra's algorithm execution time in seconds.

Number of vertices	Dijkstra's algorithm	Dijkstra's algorithm by adjacency matrix
50	0.173	0.297
100	0.322	0.329
250	0.327	0.491
500	0.334	0.4
750	0.507	0.61
1000	1.3	0.494
2500	13.425	2.223
5000	16.682	2.959
7500	34.61	6.198
10 000	81.373	24.103

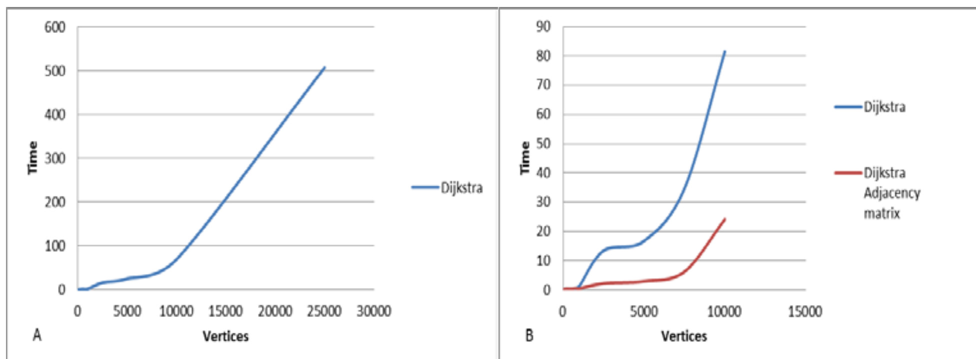

Fig. 2. Relationship between the number of vertices and execution time for Dijkstra's algorithm and its implementation: A) without adjacency matrix in C #; B - with adjacency matrix in C #.

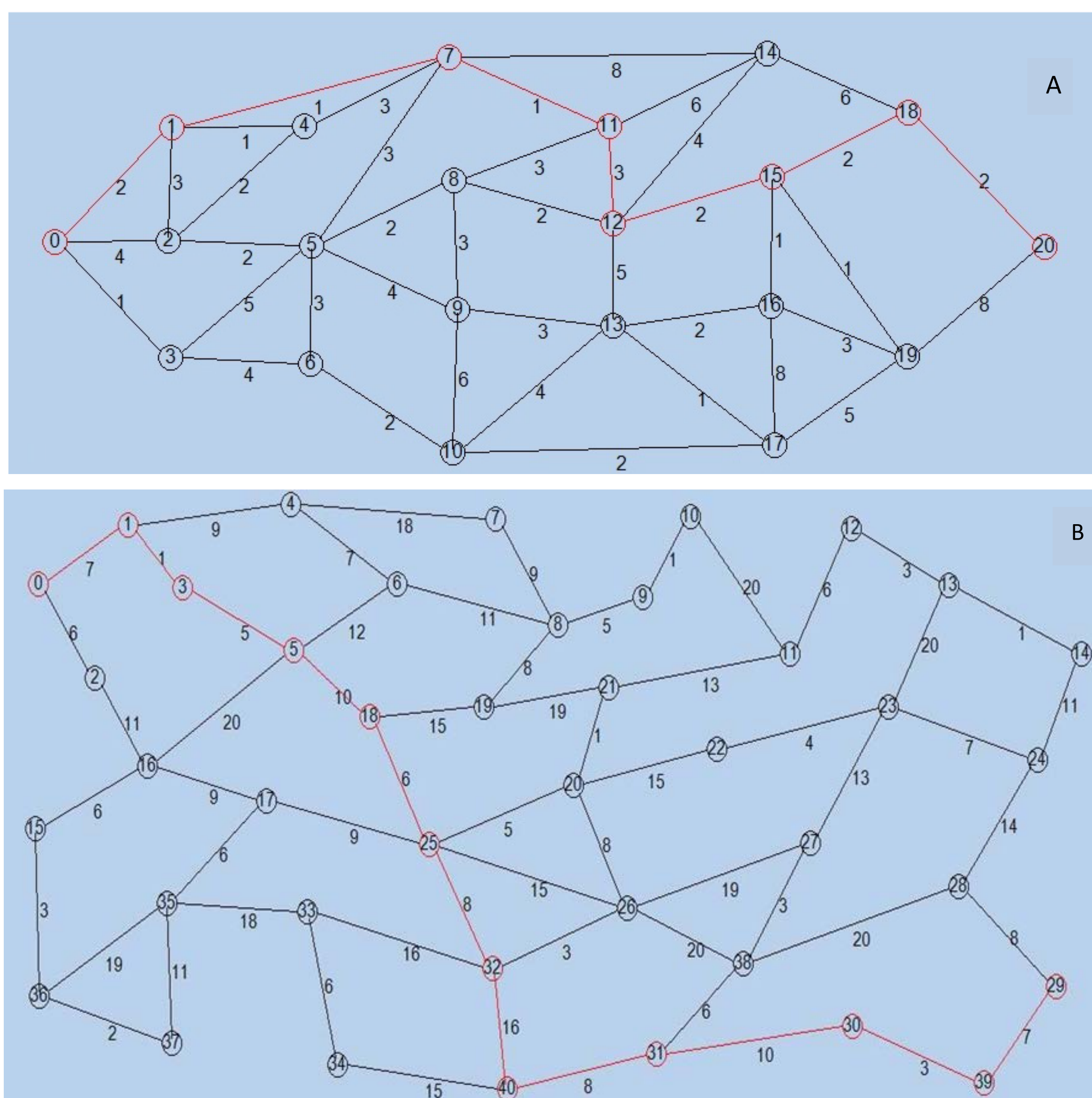

Fig. 3. Illustration of Dijkstra's algorithm search for finding shortest path from a start node to a goal node: A) n = 21, m= 41, B) n = 41, m = 60. The found shortest path is coloured in red.

In conclusion, we can say that the calculation of shortest paths in interaction graphs is an important method for network analysis in computational biology. This report draws attention to the important computational problem and provides a number of new algorithms, partially specifically tailored for biological interaction graphs.

REFERENCES

1. K. Magzhan, H. Jani, *IJSTR*, **2** (6), 100 (2013).
2. G. Pavlopoulos, M.Secrier, C.Moschopoulos, T. Soldatos, S. Kossida, J. Aerts, R.Schneider, P. Bagos *BioData Mining,* **4**(10), 1 (2011).
3. L. Ulrich, *Nucleic Acids Res.,* **35**, 386 (2007).
4. M. Krull, N. Voss, C.Choi, S. Pistor, A. Potapov, E. Wingender, *Nucleic Acids Res.*, **31**(1), 97 (2003).
5. T. Cormen, Ch. Leiserson, R. Rivest, C.Stein, Cambridge, Massachusetts 02142, *The MIT Press,* (2009).
6. W. Huber, V. Carey, L. Long, S. Falcon, R. Gentleman, *BMC Bioinformatics*, **8,** S8 (2007).
7. M. Negnevitsky, *Artificial Intelligence: A Guide to Intelligent Systems*, Third ed., Addison-Wesley, (2011).
8. J. Edmonds, *Lectures in Applied Mathematics*, **2,** 346 (1968).
9. E. Dijkstra, *Numer. Math.,* **1**, 269 (1955).
10. L. Ford, *Raud Corporation Report*, P-923 (1946).
11. R. Floyd, *Comm. ACM*, **5**, 345 (1962)
12. G. Dantzig, International Symposium, Rome, Gordon and Breach, 91, 1966.
13. B. Johnson, *Professional Visual Studio* (2015).
14. V. Vladimirov, F. Sapundzhi, R. Kraleva, V. Kralev, *Biomath Communications*, **3** (1), P71, (2016).
15. F. Sapundzhi, T.Dzimbova, N.Pencheva,P.Milanov, *Journal of Computational Methods in Molecular Design*, **5**, 98 (2015).
16. V. Kralev, *IJASEIT,* **7** (5), 1685 (2017).
17. V. Kralev, R. Kraleva, *IJACR,* **7** (28), 1 (2017).