Research Article

Aleksandar Jovanović, Ana Uzelac, Katarina Kukić*, and Dušan Teodorović

The shortest-path and bee colony optimization algorithms for traffic control at single intersection with NetworkX application

https://doi.org/10.1515/dema-2023-0160
received April 19, 2023; accepted February 9, 2024

Abstract: In this article, we study the application of NetworkX, a Python library for dealing with traffic networks, to the problem of signal optimization at a single intersection. We use the shortest-path algorithms such as Bellman-Ford (Dynamic Programming), A star (A*), and Dijkstra's algorithm to compute an optimal solution to the problem. We consider both undersaturated and oversaturated traffic conditions. The results show that we find optimal results with short Central Processor Unit (CPU) time using all the applied algorithms, where Dijkstra's algorithm slightly outperformed others. Moreover, we show that bee colony optimization can find the optimal solution for all tested problems with different degrees of computational complexity for less CPU time, which is a new contribution to knowledge in this field.

Keywords: single intersection, shortest path, optimization

MSC 2020: 68T20, 68V35, 90C35, 90C39

1 Introduction

The shortest-path problem is a problem of finding the shortest path from a starting point to a final destination. Shortest-path algorithms are mainly divided into two groups: single-source shortest paths (SSSP) and all-pairs shortest paths (APSP). The goal of the SSSP is to find the shortest path from a source node to all other nodes in the graph, while APSP aims to find the shortest paths between each pair of the nodes in the graph. In this article, the shortest-path solution algorithms are used to solve the problem of controlling signal intersection. Recall that a single intersection can be controlled using a varying number of phases. An increase in the number of phases results in greater combinatorial complexity of the problem.

In recent decades, there has been a vast interest in researching the problem of optimizing the control of a single intersection. The total vehicle delay, the total number of stopped vehicles, and the total throughput represent the usual criteria functions for optimization. Since the 1970s, by applying mathematics and numerical methods to complex transportation engineering problems, various strategies have been developed to find optimal solutions in this area. The development of one of the first software packages aimed at controlling a single intersection can be found in [1]. One of the pioneer's works considering both public transport vehicles and pedestrian flows in optimizing the control of a single intersection is [2]. The work of Pappis and Mamdani

*** Corresponding author: Katarina Kukić,** Faculty of Transport and Traffic Engineering, University of Belgrade, Vojvode Stepe 305, 11010 Belgrade, Serbia, e-mail: k.kukic@sf.bg.ac.rs
Aleksandar Jovanović: Faculty of Engineering, University of Kragujevac, Sestre Janjic 6, 34000 Kragujevac, Serbia, e-mail: a.jovanovic@kg.ac.rs
Ana Uzelac: Faculty of Transport and Traffic Engineering, University of Belgrade, Vojvode Stepe 305, 11010 Belgrade, Serbia, e-mail: ana.uzelac@sf.bg.ac.rs
Dušan Teodorović: Serbian Academy of Sciences and Arts, Knez Mihailova 35, 11000 Belgrade, Serbia, e-mail: duteodor@gmail.com

[3], in which the problem of a single intersection was solved using fuzzy logic, is considered to be one of the first applications of artificial intelligence (AI) in traffic engineering.

In [4], a simulation approach is developed to find the values of the control parameters at a single intersection for both undersaturated and oversaturated demand. The application of Markov chains to optimize the control at a single intersection was presented in [5], while genetic algorithms were used to solve the same problem in [6]. The solution of a similar traveling salesman problem by the Hopfield-type neural network was explored in [7]. Technological development has promoted the application of various AI techniques in the field of single intersection control, which led to numerous research articles, among which we highlight [8–10]. In [11], a predictive controller was developed for optimal green time balancing considering short-term traffic demand prediction. The research showed a significant reduction in queue lengths compared to traditional control logic. In [12–14], etc., different options for isolated intersection control in the presence of oversaturated traffic flows have been studied.

In this study, we present a new approach of solving the single intersection timing optimization problem by using different algorithms to solve the shortest-path problem. One of the ways to solve the shortest-path problem is to use graphs. Recall briefly that a graph, as an abstract mathematical object, is a collection of points (nodes) and lines connecting some of the points (edges). A graph is called a directed or undirected graph if one can walk along the edges on both sides or only on one side of the graph. The length of the edges is often referred to as weights and is used to calculate the shortest-path from one point to another. To compare algorithms for the shortest-path problem, we used Python and its libraries, focusing on application of the NetworkX [15]. NetworkX is a Python package publicly released in April 2005 that is used to create, manipulate, and study the structure, dynamics, and functions of complex networks. In this study, we used the latest version 2.6, released in July 2021.

Recall that nodes can be any hashable Python object, edges can contain arbitrary data represented as tuples. The library includes generators for many classical graphs and random graph models, as well as various graph algorithms for finding the shortest paths in graphs. Modeling and analyzing network data and testing new algorithms or network metrics are presented in [16]. There, the authors showed how NetworkX, in conjunction with Python packages such as SciPy, NumPy, and Matplotlib, and interfacing with other tools written in FORTRAN and C, provides a powerful tool for computational network analysis. NetworkX provides classes for representing directed and undirected graphs with optional weights and self-loops, as well as multigraphs that allow multiple edges between pairs of nodes. Adding or removing nodes and edges can be done via class methods. Graphs can be represented in three ways: as a list of edges, an adjacency matrix, or an adjacency list. Since NetworkX is easy to install and use, it has positioned itself as a powerful tool for analyzing complex networks. Numerous applications of NetworkX can be found in many research articles on various problems in engineering, epidemiology, sociology, chemistry, and other fields. Selected articles representing various applications of NetworkX are listed in Table 1.

Table 1: Selected articles with NetworkX applications

Article	NetworkX application – area and main results
[17]	The SAGE open-source mathematics system has incorporated NetworkX and extended it with more graphtheoretical algorithms and functions
[18]	An application of NetworkX in an investigation of connectivity and profits of airline companies. Graph analytics technique for the optimum solution was used for analyzing complex multigraph airlines
[19]	A novel time series classification algorithm based on complex network topology features is presented NetworkX was applied to calculate the density, modularity, average degree assortativity coefficient, etc.
[20]	A comparative analysis of four social network analysis tools – Networkx, Gephi, Pajek, IGraph based on platform, execution time, graph types, algorithms complexity, input file format, and graph features
[21]	Authors generate an undirected multigraph in NetworkX for the analysis of co-authorship of scientific documents published on malaria
[22]	A MeVer NetworkX tool is proposed, designed for analyzing and visualizing social media conversations
[23]	A hybrid data structure for storing temporal networks is implemented in DyNetworkX

As far as the authors are aware, the NetworkX package has not yet been used to solve optimization problems at a single intersection. This also represents the initial step for further application of this software package to traffic-related problems. Our motive for this work is to investigate the Central Processor Unit (CPU) time required to solve this optimization problem with different shortest-path algorithms and additionally to compare the efficiency of NetworkX with the well-known bee colony optimization (BCO), a technique that has already been used to solve this problem in [24]. In order to make this comparison, a Python code for the BCO algorithm is created. Dynamic programming (DP) is the method with optimal structure. In other words, it provides an optimal solution to a given problem. In the 2017 work [24], the authors showed that DP succeeded in solving the optimization problem for a single intersection with five phases in a CPU time of 29,120 s. In the case of control with six phases, DP was not able to solve the problem in a reasonable CPU time at that time. Therefore, the application of the BCO meta-heuristic method was justified.

The results show that NetworkX is a very useful package to solve computationally complex problems. An additional value is its accessibility and open-source concept, which makes it a highly desirable environment for solving the shortest-path problems. We emphasize that in practice, it is much easier for engineers to implement the shortest-path algorithms in such environments than to use complicated metaheuristic approaches such as BCO. Moreover, we develop the code of the BCO algorithm in the same environment, which seems to be more powerful in terms of CPU time. The optimal solution to the considered problem is provided by the present DP method, even in the complex case of traffic demand oversaturation with six-level control. Thus, we can compare the BCO results with the optimal result in the case of six-level control (with and without oversaturation).

Novelty of this study could be summarized in the following:

(1) We prove that BCO (as a swarm metaheuristic algorithm) can find the optimal solutions for six-phase single traffic control with oversaturated traffic conditions using NetworkX.
(2) Using NetworkX, in the case of optimal control at a single intersection with more than four phases, we have achieved a significantly lower CPU time than is known from the current state-of-the-art.
(3) We compared different algorithms for the shortest-path in the case of controlling a single intersection and found Dijkstra's algorithm to be the most efficient.
(4) This study serves as a comprehensive guide for those venturing into traffic network optimization. It elucidates commonly employed classical algorithms and delves into the advanced BCO technique, accompanied by pertinent pseudocodes.

This article is organized as follows. Section 2 describes the problem. Section 3 presents the proposed methodology with brief explanations and pseudocodes of the algorithms used. An experimental evaluation of the proposed approach is presented in Section 4. Recommendations for future research and conclusions are given in Section 5.

2 Statement of the problem of optimal control of single intersection

In this section, we briefly explain the pre-timed optimization of a single intersection with a given number of phases. The objective of traffic control at a single intersection is to find the optimal values for cycle length and split of green time of phases. Note that cycle length represents the execution time of all phases, to which the so-called red times are added (this is the time between two initializations of the same phase of the signal plan). A typical layout of a single intersection with associated phases is shown in Figure 1, which presents five different ways to execute traffic control in two to six phases. In Figure 1, a brief explanation of traffic control is provided by abbreviations on the left side of the figure. The simplest two-phase control is implemented by a scheme where phase 1 gives a green light for the north–south direction and phase 2 for a green light for the west–east (WE) direction. The next option is to control in three phases, where phase 1 is for the green light for the

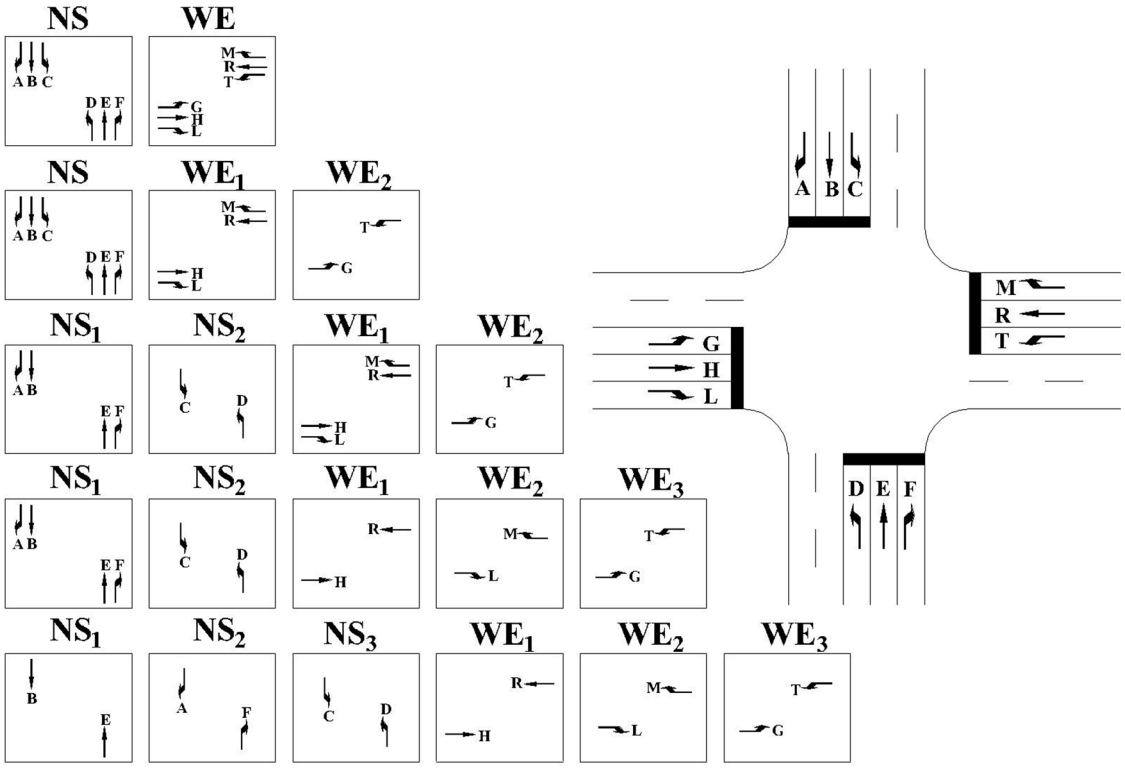

Figure 1: Layout of the single intersection with phases.

direction NS, while WE is controlled in two new phases, phase 2 for the green light on lanes H, L, R, and M and phase 3 for the green light for left turns G and T, similarly for phases 4, 5, and 6.

The problem we deal with in this article could be formulated as follows: for a given number of phases, determine optimal values for the cycle length and green times to minimize the average control delay for all vehicles arriving at the intersection within a predefined period. Furthermore, in this work, we consider both undersaturated and oversaturated traffic conditions. Oversaturated traffic conditions assume that there is no traffic demand after an analysis period (T).

2.1 Definition of the criteria function

If we denote by i the lane index, $i = 1, 2, \ldots, K$ (in Figure 1, lane indexes A, B, C, ... correspond to $i = 1, 2, 3, \ldots$), the optimization criterion is the average control delay per vehicle that occurs during the period of analysis T (which is 1 h in this case) of vehicles arriving (d_i) and is defined as follows [25]:

$$d_i = d_{1i} + d_{2i} + d_{3i}. \tag{1}$$

Note that d_{1i} represents the uniform delay per vehicle in the ith lane expressed in seconds per vehicle (s/veh), d_{2i} indicates the incremental delay per vehicle in the ith lane (s/veh), and d_{3i} stands for the initial queue delay per vehicle in the ith lane (s/veh). Mentioned delays are calculated as follows:

$$d_{1i} = 0.5 \cdot C \cdot \left(1 - \frac{g_i}{C}\right) \cdot \frac{t_i}{T} + \frac{0.5 \cdot C \cdot \left(1 - \frac{g_i}{C}\right)^2}{1 - \min\{1, X_i\} \cdot \frac{g_i}{C}} \cdot \frac{T - t_i}{T}, \tag{2}$$

$$d_{2i} = 900 \cdot T \cdot \left[(X_i - 1) + \sqrt{(X_i - 1)^2 + \frac{4X_i}{c_i \cdot T}}\right], \tag{3}$$

$$d_{3i} = \frac{1{,}800 \cdot Q_{bi} \cdot (1 + u_i) \cdot t_i}{c_i \cdot T}. \tag{4}$$

Other labels used in (2)–(4) are as follows: C is a cycle length in seconds; g_i is the green time in the ith lane (veh/h); T is the analysis period duration (1 h), and Q_{bi} is the initial queue at the start of period T. Furthermore, capacity of the ith lane (c_i), in veh/h, can be calculated as:

$$c_i = s_i \cdot \frac{g_i}{C}, \tag{5}$$

where s_i denotes the saturation flow in the ith lane (veh/h). The saturation flow rate of a traffic lane is determined by the maximum number of vehicles that could be served during one hour of green.

Volume-to-capacity ratio (X_i) can be calculated as:

$$X_i = \frac{q_i}{c_i} = \frac{q_i}{s_i \cdot \frac{g_i}{C}} = \frac{q_i \cdot C}{s_i \cdot g_i}. \tag{6}$$

Duration of unmet demand in T is denoted by t_i (h) and can be calculated as:

$$t_i = \min\left\{T, \frac{Q_{bi}}{c_i \cdot [1 - \min\{1, X_i\}]}\right\}. \tag{7}$$

Finally, u_i is the delay parameter in the ith lane and can be calculated as:

$$u_i = \begin{cases} 0, & \text{if } t_i < T, \\ 1 - \dfrac{c_i \cdot T \cdot [1 - \min\{1, X_i\}]}{Q_{bi}}, & \text{elsewhere}. \end{cases} \tag{8}$$

2.2 Mathematical formulation of the problem

If we denote by j the phase index, $j \in \{2, 3, \ldots, 6\}$, or theoretically $j \in F = \{1, 2, \ldots, f\}$, the following optimization problem can be defined as:

$$\text{Minimize} \sum_{i=1}^{K} d_i, \quad \text{subject to}: \tag{9}$$

$$\max\{C_{\min}, |F| \cdot g_{\min} + L\} \le C \le C_{\max}, \tag{10}$$

where for all $j \in F$ following is satisfied:

$$g_{\min} \le g_j \le C - L - (|F| - 1) \cdot g_{\min} \tag{11}$$

and

$$\sum_{j=1}^{|F|} g_j = C - L, \tag{12}$$

where L is all-red time (s), $g_{\min}$ is the minimum green time (s), $C_{\min}$ is the minimum allowed cycle length (s), and $C_{\max}$ is the maximum allowed cycle length (s).

The objective function (9) to be minimized represents the average control delay experienced by all vehicles arriving at the intersection within a given period. Equation (10) defines the integer interval of possible cycle length values, while equation (11) defines the integer interval of possible green time values. The relationship between cycle length, green time, and all-red time is described by equation (12).

3 Methodology

A main component of this approach for controlling individual intersections is the creation of a network in the Python environment that contains nodes and edges. Each node represents a value from the range defined in (11), while the edges represent the corresponding values of the criterion function obtained from equations (1)–(8). Then, using NetworkX, the Python library for network processing, we applied algorithms to find the optimal shortest-path, such as Dijkstra, Dynamic Programming, and A*. We also compare the optimal solutions obtained using the NetworkX package with those obtained using BCO solutions obtained, also programmed in the Python environment. That way, we obtain additional contribution of this study – the validation that BCO consistently determines optimal values for both cycle length and green times across varying numbers of phases.

3.1 Network building for the application of the shortest-path algorithms

The network is divided into a series of layers, each layer (except the first) consisting of all allowed green time values for that phase. The first layer contains the cycle length value minus all-red time, while the last layer contains the sum of all green times. Thus, the first and last layers each consist of only one node with the same value.

Let us show the structure of the sub-network in Figure 2, for the case of four phases with a cycle length of $C = 64$ s, the all-red time of $L = 14$ s and the minimum green time of $g_{\min} = 5$ s.

The sub-network of Figure 2 is an oriented graph in which the nodes are connected only between layers, while the edges in the layer itself do not exist. Furthermore, the layers are conditionally connected as the $g_{\min}$ value is maintained. For instance, the nodes of layer g_1 and the nodes of layer $g_1 + g_2$ are connected if and only if:

$$\text{node\_value}(g_1 + g_2) - \text{node\_value}(g_1) \geq g_{\min}. \tag{13}$$

Exceptions are the first and the last layers, which are completely connected with their neighboring layers. The values of the green times within each layer are determined by the sum of the green times of all previous layers. Thus, the minimum green time for these layers is subject to $g_{\min}$, as shown in Figure 2. The maximum green time for layers g_1, $g_1 + g_2$, and $g_1 + g_2 + g_3$ are, respectively, equal to:

$$C - L - (|F| - 1)\cdot g_{\min},\quad C - L - (|F| - 2)\cdot g_{\min},\quad \text{and}\quad C - L - (|F| - 3)\cdot g_{\min}. \tag{14}$$

The weights of network edges between layers i and j (w_{i-j}) represent the values of the criterion function. For example, the weight between node 5 (layer g_1) and node 10 (layer $g_1 + g_2$) is equal to (Figure 1):

$$w_{5-10} = d_C + d_D, \tag{15}$$

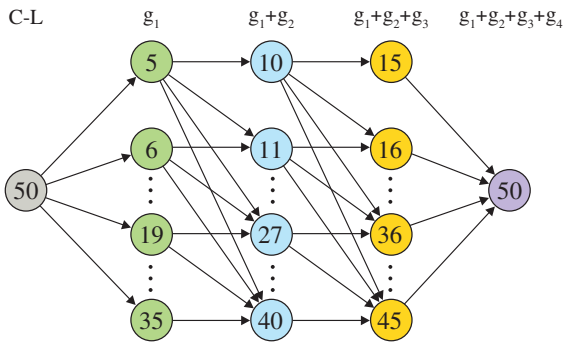

Figure 2: An example of the corresponding sub-network design with four phases.

where d_C and d_D take corresponding values from (1). The number of sub-networks corresponds to the number of possible cycle length values determined by (10). We solve each sub-network, with the global optimal solution corresponding to the minimum value of the objective function resulting from all sub-networks. The pseudo-code for the building of the corresponding network is presented as follows:

Pseudocode for building of the corresponding network:

Set number of phases: nop
Set initial values for minimum allowed cycle length: $C_{\min}$
Set initial values for maximum allowed cycle length: $C_{\max}$
Set initial value for minimal green time: $Z_{\min}$
Set initial value for lost time (all red time): L
For $C = C_{\min}$ to $C_{\max} + 1$
  For $i = 1$ to nop
    Set initial traffic demands q_i as list of values
    Set saturation flows s_i as list of values
    Define Z as list of values from $Z_{\min}$ to $C - L - (nop - 1) \cdot Z_{\min} + 1$
    Define capacity function $k_i(s_i, Z, C)$ to return $s_i \cdot (Z/C)$
    Define volume-to-capacity ratio $X_i(q_i, k_i)$ to return q_i/k_i
    Define control delay by phases function $d_i(Z, C, k_i, X_i)$ to
    return $\frac{0.5 \cdot C \cdot \left(1 - \frac{Z}{C}\right)^2}{1 - \min\{1, X_i\} \cdot \frac{Z}{C}} + 900 \cdot \left[(X_i - 1) + \sqrt{(X_i - 1)^2 + \frac{4X_i}{k_i}}\right]$
    Define function D_i for each phase
  Initialize bidirectional Graph G
  Add node $(C - L)$ to the Graph G
  For $i = 1$ to $nop - 1$
    Create list of nodes for each cycle
    Add nodes to Graph G
    Calculate D function for each pair of neighboring nodes
    Add edges between neighboring nodes and set distance D
  Call shortest-path algorithm
  Save values returned by algorithm
Find minimum values of the algorithm returned values
Print optimal values for cycle length and green times

3.2 Solution algorithms

Regarding the existing literature, there are a certain number of articles in which authors discussed and compared the results received from using shortest-path algorithms, or tried to short their execution time, etc. For example, authors in [26] presented a new hybrid algorithm called Bellman-Ford-Dijkstra by combining Bellman-Ford and Dijkstra algorithms with the proof that the new algorithm can generate the shortest-path tree. In [27], Dijkstra and Bellman-Ford algorithms were compared. The authors tested the execution time of these algorithms for a different number of nodes and concluded that when the number of nodes is small, the running time of Bellman-Ford algorithm is better than Dijkstra's. If a graph has a greater number of nodes, the results show that Dijkstra's algorithm has a lower execution time. For real-time applications, the authors give an advantage to the Dijkstra's. Similar results were obtained in [28], where authors show that the Bellman-Ford algorithm is slightly superior in the case of a small number of nodes, while Dijkstra is more effective for a large number of nodes.

The main novel contribution of this article in this field is a new approach to address the pre-timed optimization problem at a single intersection by using different shortest-path algorithms. As far as authors know, NetworkX package has not been used so far to deal with optimization problem at single intersection. Our motif for this study is to explore the CPU time required to solve this optimization problem, and to compare the efficiency of NetworkX with the well-known BCO metaheuristic approach that has already been used to solve the problem.

To make this article self-readable, we briefly explain the ideas behind algorithms used in the analysis and present pseudocode for each of them.

3.2.1 Dijkstra's algorithm

The original algorithm, published in 1959 in [29], contains a set of approximations for the shortest paths from one source to the other nodes and improves these approximations in each iteration of the algorithm. To this end, it defines two sets of nodes, one for which the shortest-path problem has already been solved and one that contains all other nodes. In the first step of Dijkstra's algorithm [30], the distance for the starting point is set to 0 and the distance for all other points is set to infinity. Second, all points (including the starting point) are set as unvisited nodes. Third, the unvisited node with the smallest current distance is set as the current node C. The further procedure is applied for each neighbor N of the current node: the current distance of C is added with the weight of the edge connecting C–N, and if it is smaller than the current distance of N, it is set as the new current distance of N. After that, the current distance of N is set as the new current distance of C. Finally, the current node C is marked as visited and the previous procedure is repeated until the target point is visited.

Pseudocode for Dijsktra's Algorithm for graph G:

```
for c in G:
    distance(c)=∞
    visited(c)=0
distance(n_start)=0
do while visited(n_goal)==0
REPEAT
    min_dist=∞, currentNode=G[0]
    for c in G:
        if min_dist > distance(c) then
            min_dist=distance(c), currentNode=c
    for c in G:
        if visited(c)==0 AND (c is neighbor of currentNode) then
            if (distance(currentNode)+edge_distance(c,currentNode)) < distance(c)
            distance(c)=distance(currentNode)+edge_distance(c,currentNode)
    visited(currentNode)=1
UNTIL visited(n_goal)==1
```

The time complexity of the Dijkstra's algorithm is $O(n^2)$ and with the use of a Fibonacci heap can be reduced to $O((n + E)\cdot \log n)$ [31,32], where n represents the number of nodes, while E is the number of edges.

3.2.2 A* Algorithm

A* algorithm was first published in 1968 [33]. Briefly, it is working the following way: it generates a tree of paths from the start node and extends them by one edge at a time until it reaches the goal node or if there are no paths suitable to be extended.

Pseudocode for A* Algorithm:

```
for c in G:
    distance(c)=∞
    visited(c)=0
distance(n_start)=0, condition = TRUE
do while visited(n_goal)==0
REPEAT
   min_dist=∞
    currentNode=G[0]
    for c in G:
        if min_dist > distance(c) then
            min_dist=distance(c)
            currentNode=c
    for c in G:
        if c==n_goal then condition = FALSE
        if visited(c)==0 AND (c is neighbor of currentNode) then
            temp_distance=distance(currentNode)+edge_distance(c,currentNode)+heuristic(c)
            if temp_distance < distance(c)
                distance(c)=temp_distance
    visited(currentNode)=1
UNTIL condition
```

Based on the estimate of the cost of the path, A* determines which path to extend at each iteration. More formally, it follows the same steps as described in Dijkstra's algorithm with the exception. Heuristic is added to the third step, which can be described as, [30]: for each neighbor N of the current node, add the current distance of C with the weight of the edge connecting C–N and the weight to the destination point (heuristic). If it is smaller than the current distance of N, its new value is the new current distance of N. The remaining steps stay the same as in Dijkstra's algorithm.

The time complexity of A* is $O(b^d)$, where d represents the depth of the solution and b is a branching factor (the average number of successors per state). It can also be interpreted as $O((n + E)\cdot \log(n))$, where E represents the number of edges and n represents a number of nodes [34].

3.2.3 Bellman-Ford's algorithm

First published in [35] and [36], Bellman-Ford's algorithm is based on the idea that there can be at most $n - 1$ edges in one of the paths from the starting node to any other node in the graph, where n represents the number of nodes in the graph.

Pseudocode for Bellman-Ford's:

```
for c in G:
    distance(c)=∞, previous(c)=null
distance(c_start) = 0
for c in G:
    for edge (c, c′) in G:
        if (distance(c′)+edge_weight(c′, c))<distance(c) then
            distance(c)= distance(c′)+edge_weight(c′, c)
            previous(c)=c′
```

Therefore, if we iterate $n - 1$ times, we are guaranteed to find the shortest-path from source to goal. Bellman-Ford updates along all edges for every iteration – it examines each edge if it lessens the shortest-path distance. At the beginning, Bellman-Ford's algorithm initializes the distance from the source to all other nodes to infinity. Then, if the distance to the destination can be shortened by taken the edge, the distance values are updated to the new lower value. It searches the structure of the graph and generates a better solution.

The time complexity of this algorithm is $O(n \cdot E)$, where n represents the number of nodes, while E represents a number of edges, [37].

3.2.4 BCO

The BCO algorithm is part of the swarm optimization metaheuristic and was originally proposed by [38] and [39]. BCO seeks feasible solutions to difficult combinatorial optimization problems by mimicking the behavior of natural bees in nectar collection.

In this study, we apply the improved version of the BCO algorithm with the following parameters: B, the number of bees involved in the search; IT, the number of iterations; NP, the number of forward and backward passes in a single iteration; NC, the number of changes in a forward pass; and S, the best-known solution. The pseudo-code of the BCO algorithm [40] is as follows:

Pseudocode for BCO:

```
Set number of bees: B
Set the number of iteration: IT
Set the number of forward and backward passes in a single iteration: NP
Set number of changes: NC
REPEAT
    For i = 1 to B
        Determine an initial solution for the ith bee
        Evaluate the solution of the ith bee
    Set S: calculate the best solution of the bees
    For j = 1 to IT:
        For i = 1 to B:
            Set the initial solution for bee i
        For k = 1 to NP:
            For i = 1 to B:
                For r = 1 to NC:
                    Evaluate modified solutions generated by possible changes of the ith bee solution
                    By roulette wheel selection choose one of the modified solutions
```

For i = 1 to B:
    Evaluate solution of the ith bee
For i = 1 to B:
    Make a decision whether the ith bee is loyal
For i = 1 to B:
    if the bee i is not loyal
        Choose one of the loyal bees to be followed by the ith bee
    if the best solution of the bees is better then the solution S
        Set S: the best solution of the bees
UNTIL criteria is met
RETURN S

More details about the BCO algorithm and some practical applications can be found in the handbook [41]. The BCO algorithm proves to be a useful tool for solving the problem of traffic control at a single intersection. In [24], the authors have shown that BCO can outperform genetic algorithms and provide optimal solutions. In that article, optimality is demonstrated using a dynamic programming approach, which was, however, very time-consuming due to the technical characteristics of the available computers at that time. The BCO approach to control individual intersections is described in detail in articles [42,43]. In this article, BCO is used as a comparison tool for the shortest-path algorithms.

For this purpose, we rewrote the original BCO code from Java to Python. In the beginning, we assign the initial solution and compute its fitness function. Then, we go through iterations to find the smaller fitness function. The rest of the code is divided into two phases: the forward phase and the backward phase. The BCO algorithm goes through iteration after iteration until a stopping condition is met. Our stopping condition was minimal CPU time achieved by the shortest-path algorithms. When the stopping condition is satisfied, the current best solution is reported as the final solution. We compared this solution to the solution obtained by the other shortest-path algorithms.

4 Numerical example

The proposed algorithms are tested using the layout in Figure 1 as an example. We have analyzed the cases where the total number of phases is 2, 3, 4, 5, and 6, respectively. All-red times increase with the number of phases, from 10 s for 2 phases to 18 s to oversaturated scenario with 6 phases, with a step size of 2 s. The minimum allowable cycle length $C_{\min}$ and the maximum allowable cycle length $C_{\max}$ are 30 and 120 s, respectively, while the minimum value for the green time $g_{\min}$ is 5. The analyzed period is set to 1 h (T = 1 h). The traffic demands (q_i in veh/h) and the saturation flows (s_i in veh/h) used for the calculations are given in Tables 2 and 3, respectively.

Note that values for initial queue Q_{bi} are assignable only in the case of 6-phase control, just because this is the most complex case from the perspective of CPU time usage. By introducing the initial queue, our aim was to further intricate Formula (1) in this most complex scenario and assess its impact on the code execution CPU time.

Table 2: Traffic demands data per lane

Traffic demands	Lane											
	A	B	C	D	E	F	G	H	L	M	R	T
q_i	210	315	145	190	290	180	175	300	185	200	320	165
Q_{bi}	0	30	0	0	27	0	0	58	0	0	14	0

Table 3: Saturation flow data per lane and per number of phases

	Lane											
Number of phases	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**L**	**M**	**R**	**T**
2	1,500	1,600	1,350	1,300	1,600	1,500	1,400	1,600	1,500	1,500	1,600	1,300
3	1,500	1,600	1,350	1,300	1,600	1,500	1,500	1,900	1,500	1,500	1,900	1,500
4–6	1,500	1,900	1,500	1,500	1,900	1,500	1,500	1,900	1,500	1,500	1,900	1,500

In the implementation of the proposed BCO algorithm, we set $B = 20$, $NP = 20$, $NC = 1$. These values were chosen based on previous experience with similar problems. In this article, the stopping criteria for BCO are the shortest CPU time achieved among all other tested algorithms.

4.1 Experimental results

In this subsection, we present experimental results. All experiments were initially performed on a PC with a 2.7 GHz Intel Core i7-6828HQ processor and 32 GB RAM under MS Windows OS. This configuration represents a performance standard accessible to home users, which motivated our choice for the study. However, to validate these results, we also executed the Python codes on a faster computer to determine if the computer's performance would influence the execution speed ranking of the tested algorithms. Those validated results are presented in the Appendix.

To evaluate the performance of three considered algorithms (Dijkstra, Bellman-Ford's, and A*), we executed a Python code written using the NetworkX library ten times for each algorithm. With conditions described in Tables 2 and 3 (the traffic demands and the saturation flows), each algorithm provided the optimal solution for cycle length and green times (given in seconds) that minimized the average control delay experienced by all vehicles arriving at the intersection within a given period. These optimal solutions and corresponding minimal delays are given in Table 4.

Recall that the relationship between cycle length, green time, and all-red time is described by equation (12), so the cycle length presented in Table 4 is obtained as the sum of green times given below corresponding cycle and all-red time, which is equal to 10 s for two phases to 18 s for six phases.

Mean values and standard deviations (STD) of CPU times (in seconds) for performed ten tests for each of the three tested shortest-path algorithms are presented in Table 5. All tests are carried out for the total number of phases equal to 2, 3, 4, 5, 6 as well as for an oversaturated scenario with six phases.

Obtained results showed that A* and Dijkstra's algorithm have similar performance, but slight advantage may be given to Dijkstra's algorithm due to lower values of CPU time for all scenarios except with five phases, as well as lower standard deviations that are observed for Dijkstra's algorithm with six phases, which is probably caused by the heuristic in A*. CPU times for Bellman-Ford's algorithm are slightly larger than for the other two algorithms and that trend becomes more visible by the increased number of phases (Figure 3).

Table 4: Optimal values for the cycle length, green times, and delay

	Number of phases					
	2	**3**	**4**	**5**	**6**	**6 o.s.**
Cycle length	32	41	56	90	139	140
Green times	11, 11	12, 10, 7	12, 9, 12, 9	17, 13, 17, 14, 13	23, 20, 18, 24, 19, 17	26, 19, 17, 26, 18, 16
Delay	131.37	240.74	409.70	759.36	1576.91	2327.77

Table 5: Mean values and standards deviations of CPU times (performed on a PC with a 2.7 GHz Intel Core i7-6828HQ processor and 32 GB RAM under MS Windows OS)

	Number of phases					
	2	**3**	**4**	**5**	**6**	**6 o.s.**
	Dijkstra's algorithm					
Mean CPU time (s)	0.929	2.526	3.012	3.852	4.955	5.900
STD (s)	0.038	0.061	0.183	0.307	0.074	0.113
	Bellman-Ford's algorithm					
Mean CPU time (s)	0.940	2.618	3.123	3.914	5.486	6.442
STD (s)	0.019	0.117	0.123	0.077	0.138	0.188
	A* algorithm					
Mean CPU time (s)	0.930	2.620	3.048	3.797	5.134	5.929
STD (s)	0.019	0.049	0.164	0.114	0.299	0.167

Also, results from Table 5 are visually presented in Figure 4 on which mean values of CPU times in seconds are shown in the form of error bars, showing how precise a measurement is.

Based on the obtained CPU times, we set the minimum obtained time for each number of phases as the termination criterion for BCO to allow comparison. The application of this sophisticated method resulted in optimal values for cycle length and green times for all numbers of phases. The added value of this research is to continue the results obtained in [24]. The application of the shortest-path and BCO algorithms in Python and NetworkX confirmed the ability of BCO to converge to an optimal solution in a very short time, even for problems of controlling a single intersection with five and six phases, a result that we were not able to confirm a few years ago with the technical capabilities available to us. This result further confirms the application of BCO to even more complex problems, where the implementation of shortest-path algorithms is still too time-consuming.

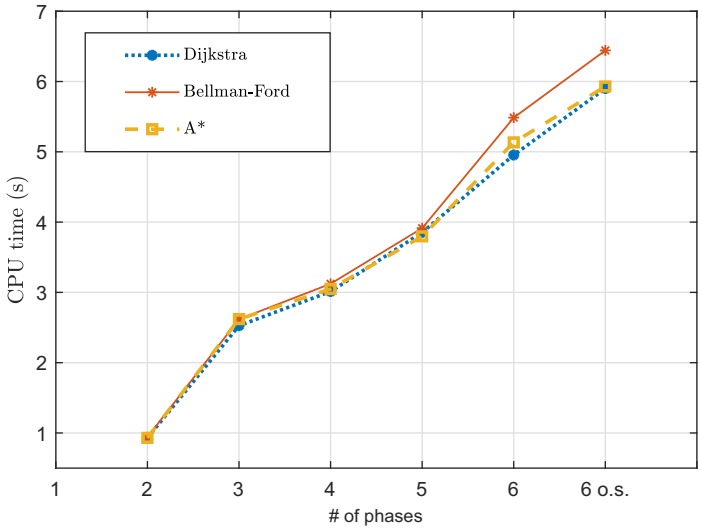

Figure 3: Mean CPU times per number of phases for tested algorithms.

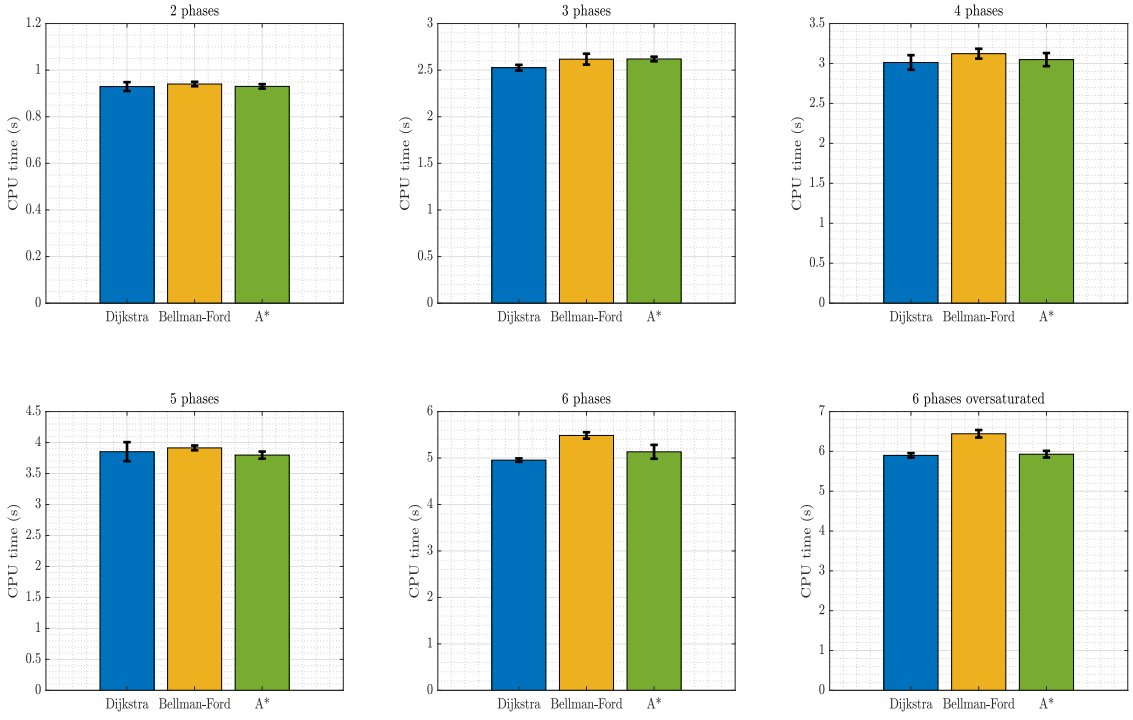

Figure 4: Mean CPU times with error per number of phases for tested algorithms.

5 Conclusion

In this study, we have proposed a comparison between the sophisticated meta-heuristic optimization method BCO and a classical shortest-path algorithms in a Python environment, specifically the NetworkX package. The implementation was motivated by a problem from traffic engineering-optimizing the control of a single intersection. Although it is a relatively simple optimization problem, until recent technological development, it was very time-consuming to apply the shortest-path algorithms in the case of six-phase control. Based on the results obtained, we see that the Dijkstra and A* algorithms slightly outperform the Bellman-Ford's algorithm for all numbers of phases, with the difference in CPU times being more pronounced for a larger number of phases. A* and Dijkstra's algorithm showed very similar CPU times, with slightly varying standard errors that are larger for Dijkstra's algorithm in the case of five phases and for A* in the case of six phases, for both undersaturation and oversaturation.

It has already been shown that BCO can find a good solution with reasonable CPU time and outperforms the genetic algorithm in the case of controlling a single signalized intersection [24]. However, in the time when that research was conducted, 2017, it was not possible to find the optimal solution for a complex control, such as oversaturated optimization with six phases. With the improvement of hardware, we found that BCO can achieve the optimal solutions in all cases of controlling a single intersection. Even more, the obtained CPU times presented in this article for the most computationally intensive tasks opens a new avenue for the application of real-time control strategies for complex intersections and coordination systems based on the achieved CPU time for an optimal solution in fixed-time optimization. One possible application is the optimization policies for adaptive control strategy proposed in [44].

One of the added values of this research is its educational purpose, as we systematize the pseudocodes of all implemented algorithms. The codes in Python are also available upon request. The authors believe that traffic engineers can benefit from this research, as well as mathematicians who can also solve the problem of controlling a single intersection by modifying the proposed codes by applying newly developed optimization techniques.

In future work, the authors hope to develop similar techniques to address optimization problems of arterial or even area-wide urban traffic and also to implement NetworkX options and capabilities to solve the problems of controlling new, alternative intersection solutions with quite complicated geometry (such as restricted crossing U-turn or diamond diverging intersections).

Funding information: The authors state that there is no funding involved.

Author contributions: All the authors contributed equally and significantly in writing this article.

Conflict of interest: The authors declare that they have no conflict of interest.

Data availability statement: The codes generated for the purpose of the current study are available from the authors on a reasonable request.

References

[1] R. E. Allsop, *Delay-minimising settings for fixed-time traffic signals at a single road junction*, J. Inst. Math. Appl. **8** (1971), no. 2, 164–185, DOI: https://doi.org/10.1093/imamat/8.2.164.
[2] K. Bang, *Optimal control of isolated traffic signals (abridgment)*, Transport. Res. Board **597** (1976), 33–35.
[3] C. P. Pappis and E. H. Mamdani, *A fuzzy logic controller for a traffic junction*, IEEE Trans. Syst. Man Cybernetics **7** (1977), no. 10, 707–717.
[4] A. A. Saka, G. Anandalingam, and N. J. Garber, *Traffic signal timing at isolated intersections using simulation optimization*, Conference on Winter Simulation, 1986. pp. 795–801.
[5] W. Brilon and N. Wu, *Delays at fixed-time traffic signals under time-dependent traffic conditions*, Traffic Eng. Control **31** (1990), no. 12, 623–631.
[6] M. D. Foy, R. F. Benekohal, and D. E. Goldberg, *Signal timing determination using genetic algorithms*, Transport. Res. Record **1365** (1992), 108.
[7] J. Mańdziuk, *Solving the travelling salesman problem with a Hopfield-type neural network*, Demonstr. Math. **29** (1996), no. 1, 219–232, DOI: https://doi.org/10.1515/dema-1996-0126.
[8] M. B. Trabia, M. S. Kaseko, and M. Ande, *A two-stage fuzzy logic controller for traffic signals*, Transport. Res. Part C **7** (1999), no. 6, 353–367.
[9] K. Dresner and P. Stone, *Traffic intersections of the future*, *Proceedings of the National Conference on* Artif. Intelligence **21** (2006), no. 2, 1593.
[10] K. S. Duisters, *Formulating and testing an algorithm for fixed time control of traffic intersections*, Internship report, Eindhoven University of Technology, 2013.
[11] S. Jafari, Z. Shahbazi, and Y. C. Byun, *Improving the performance of single-intersection urban traffic networks based on a model predictive controller*, Sustainability **13** (2021), no. 10, 5630, DOI: https://doi.org/10.3390/su13105630.
[12] T. H. Chang and J. T. Lin, *Optimal signal timing for an oversaturated intersection*, Transport. Res. Part B Methodological **34** (2000), no. 6, 471–491.
[13] F. Dion, H. Rakha, and Y. S. Kang, *Comparison of delay estimates at under-saturated and over-saturated pre-timed signalized intersections*, Transport. Res. Part B Methodological **38** (2004), no. 2, 99–122.
[14] M. Zhang and B. Lan, *Detect megaregional communities using network science analytics*, Urban Sci. **6** (2022), no. 12, 12, DOI: https://doi.org/10.3390/urbansci6010012.
[15] NetworkX, Network Analysis in Python. https://networkx.org/
[16] A. Hagberg, D. Schult, and P. Swart, *Exploring network structure, dynamics, and function using NetworkX*, In Proceedings of the 7th Python in Science Conference SciPy Conference- Pasadena, CA, August 19–24, 2008, pp. 11–16.
[17] The Sage Development Team, Graph Theory, Release 9.6, 2022, https://doc.sagemath.org/pdf/en/reference/graphs/graphs.pdf.
[18] V. V. R. Kollu, S. S. Amiripalli, M. S. N. Jitendra, and T. R. Kumar, *A network science-based performance improvement model for the airline industry using NetworkX*, Int. J. Sensors Wireless Commun. Control **11** (2021), no. 7, 768–773, DOI: https://doi.org/10.2174/2210327910999201029194155.
[19] H. Li, R. Jia, and X. Wan, *Time series classification based on complex network*, Expert Syst. Appl. **194** (2022), 116502, DOI: https://doi.org/10.1016/j.eswa.2022.116502.
[20] N. Akhtar, *Social network analysis tools*, In Proceedings 4th International Conference on Communication Systems and Network Technologies, 2014, pp. 388–392, DOI: https://doi.org/10.1109/CSNT.2014.83.

[21] R. Azondekon, Z. J. Harper, F. R. Agossa, C. M. Welzig, and S. McRoy, *Scientific authorship and collaboration network analysis on malaria research in Benin: papers indexed in the web of science (1996-2016)*, Global Health Res. Policy **3** (2018), no. 1, pp. 1–11, DOI: https://doi.org/10.1186/s41256-018-0067-x.
[22] O. Papadopoulou, T. Makedas, L. Apostolidis, F. Poldi, S. Papadopoulos, and I. Kompatsiaris, *MeVer NetworkX: Network analysis and visualization for tracing disinformation*, Future Internet **14** (2022), no. 5, 147, DOI: https://doi.org/10.3390/fi14050147.
[23] T. Hilsabeck, M. Arastuie, and K. S. Xu, *A hybrid adjacency and time-based data structure for analysis of temporal networks*, In Proceedings of the 10th International Conference on Complex Networks and Their Applications, 2022, pp. 593–604, DOI: https://doi.org/10.48550/arXiv.2206.11444.
[24] A. Jovanović and D. Teodorović, *Pre-timed control for an under-saturated and over-saturated isolated intersection: A Bee Colony Optimization approach*, Transport. Plan Technol. **40** (2017), no. 5, 556–576, DOI: https://doi.org/10.1080/03081060.2017.1314498.
[25] H. C. Manual, *HCM 2000*. Washington, DC: Transportation Research Board, 2000.
[26] Y. Dinitz and R. Itzhak, *Hybrid Bellman-Ford-Dijkstra algorithm*, J. Discrete Algorithms **42** (2017), 35–44, DOI: https://doi.org/10.1016/j.jda.2017.01.001.
[27] S. W. G. Abusalim, R. Ibrahim, M. Zainuri Saringat, S. Jamel, and J. Abdul Wahab, *Comparative analysis between Dijkstra and Bellman-Ford algorithms in shortest path optimization*, IOP Confer. Ser. Materials Sci. Eng. **917** (2020), no. 1, 012077, DOI: https://doi.org/10.1088/1757-899X/917/1/012077.
[28] J. B. Singh and R. C. Tripathi, *Investigation of Bellman-Ford Algorithm, Dijkstra's Algorithm for suitability of SPP*, Int. J. Eng. Development Res. **6** (2018), no. 1, 755–758.
[29] E. W. Dijkstra, A Note *on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), 269–271.
[30] D. Rachmawati and L. Gustin, *Analysis of Dijkstra's algorithm and A* algorithm in shortest-path problem*, J. Phys. Conference Series **1566** (2020), no. 1, 012061, DOI: https://doi.org/10.1088/1742-6596/1566/1/012061.
[31] M. Barbehenn, *A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices*, IEEE Trans. Comput. **47** (1998), no. 2, 263, DOI: https://doi.org/10.1109/12.663776.
[32] P. Sanders and D. Schultes, *Engineering fast route planning algorithms*, In: C. Demetrescu, (eds) Experimental Algorithms. WEA. Lecture Notes in Computer Science, vol. 4525, Springer, Berlin, Heidelberg, 2007.
[33] P. E. Hart, N. J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Trans. Syst. Sci. Cybernetics **4** (1968), no. 2, 100–107, DOI: https://doi.org/10.1109/TSSC.1968.300136.
[34] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Pearson, New York, USA, 2009.
[35] L. R. Ford, *Network Flow Theory*, Paper P-923. RAND Corporation, Santa Monica, California, 1956.
[36] R. Bellman, *On a routing problem*, Quart. Appl. Math. **16** (1958), 87–90.
[37] S. Lewandowski, *Shortest paths and negative cycle detection in graphs with negative weights-I: The Bellman-Ford-Moore algorithm revisited*, Comput. Sci. (2010), DOI: https://doi.org/10.18419/OPUS-2678.
[38] P. Lučić and D. Teodorović, *Bee system: Modeling combinatorial optimization transportation engineering problems by swarm intelligence*, Preprints of the TRISTAN IV Triennial Symposium on Transportation Analysis, Sao Miguel, Azores Islands, 2001, pp. 441–445.
[39] P. Lučić and D. Teodorović, *Transportation modeling: an artificial life approach*, Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, 2002, pp. 216–223.
[40] M. Nikolić and D. Teodorović, *Empirical study of the bee colony optimization (BCO) algorithm*, Expert Syst. Appl. **40** (2013), no. 11, 4609–4620, DOI: https://doi.org/10.1016/j.eswa.2013.01.063.
[41] D. Teodorović, T. Davidović, M. Šelmić, and M. Nikolić, *Bee colony optimization and its applications*, In: A. Kulkarni (Eds.), Handbook of AI-Based Metaheuristics, Routledge, Taylor & Francis Group, London, UK, 2021.
[42] A. Jovanović and D. Teodorović, *Fixed-time traffic control at superstreet intersections by bee colony optimization*, Transport. Res. Record **2676** (2022), no. 4, 228–241, DOI: https://doi.org/10.1177/03611981211058104.
[43] A. Jovanović and D. Teodorović, *Multi-objective optimization of a single intersection*, Transport. Plan. Technol. **44** (2021), no. 2, 139–159, DOI: https://doi.org/10.1080/03081060.2020.1868083.
[44] N. H. Gartner, *OPAC: a demand-responsive strategy for traffic signal control*, Transport. Res. Record **906** (1983), 75–84.

Appendix

Mean values and standard deviations (STD) of CPU times (in seconds) for performed ten tests for each of the three tested shortest-path algorithms on faster computer (PC with a 4.6 GHz AMD Ryzen 5,900 × 12-core processor and 16 GB RAM under MS Windows OS) are presented in Table A1.

Results presented in Table A1 confirm that Dijkstra's slightly outperformed other the shortest-path algorithm regarding execution speed and BCO was always able to find optimal solution in given time frame.

Table A1: Mean values and STD of CPU times (performed on a PC with a 4.6 GHz AMD Ryzen 5,900 × 12-core processor and 16 GB RAM under MS Windows OS)

	Number of phases					
	2	**3**	**4**	**5**	**6**	**6 o.s.**
	Dijkstra's algorithm					
Mean CPU time (s)	0.531	1.284	1.587	1.967	2.710	3.132
STD (s)	0.003	0.010	0.006	0.007	0.022	0.009
	Bellman-Ford's algorithm					
Mean CPU time (s)	0.539	1.318	1.647	2.115	2.897	3.353
STD (s)	0.012	0.011	0.013	0.015	0.011	0.021
	A* algorithm					
Mean CPU time (s)	0.534	1.307	1.605	1.997	2.732	3.159
STD (s)	0.004	0.017	0.019	0.008	0.017	0.009