Computing and Informatics, Vol. 38, 2019, 57–84, doi: 10.31577/cai_2019_1_57

VARIABLE NEIGHBORHOOD SEARCH APPROACH FOR SOLVING ROMAN AND WEAK ROMAN DOMINATION PROBLEMS ON GRAPHS

Marija Ivanović

Faculty of Mathematics
University of Belgrade
Studentski trg 16/IV
11 000 Belgrade, Serbia
e-mail: marijai@math.rs

Dragan Urošević

Mathematical Institute, SANU
Kneza Mihaila 36
11 000 Belgrade, Serbia
e-mail: draganu@mi.sanu.ac.rs

Abstract. In this paper Roman and weak Roman domination problems on graphs are considered. Given that both problems are NP hard, a new heuristic approach, based on a Variable Neighborhood Search (VNS), is presented. The presented algorithm is tested on instances known from the literature, with up to 600 vertices. The VNS approach is justified since it was able to achieve an optimal solution value on the majority of instances where the optimal solution value is known. Also, for the majority of instances where optimization solvers found a solution value but were unable to prove it to be optimal, the VNS algorithm achieves an even better solution value.

Keywords: Roman domination in graphs, weak Roman domination in graphs, combinatorial optimization, metaheuristic, variable neighborhood search

Mathematics Subject Classification 2010: 05C69, 05C85, 90C10

1 INTRODUCTION

The Roman domination problem (RD problem) was introduced by ReVelle and Rosing [1] and Cockayne et al. [2] and can be interpreted as follows.

Assuming that any province of the Roman Empire is considered to be safe if there is at least one legion (of maximum 2) stationed within it, the RD problem requires that every unsafe province must be adjacent to a province with at least two legions stationed within it and the total number of stationed legions within all provinces of the Roman Empire is minimal.

In a graph terminology, let $G = (V, E)$ be a simple undirected graph with a vertex set V such that each vertex $u \in V$ represents a province of the Roman Empire and each edge, $e \in E$, represents an existing connection between two provinces. Let f be a function $f : V \to \{0, 1, 2\}$ and let the weight of the vertex u, denoted by $f(u)$, represent the number of legions stationed at province u. Further, let the weight of the function f be calculated by a formula $\sum_{v \in V} f(v)$. Function f is called a Roman dominating function (RD function) if every vertex u such that $f(u) = 0$ is adjacent to a vertex v such that $f(v) = 2$. The Roman domination problem is to find an RD function f of a graph G with the smallest weight. The smallest weight of the RD function f, denoted by $\gamma_R(G)$, is known as the Roman domination number.

We illustrate the Roman domination problem in the example below.

Example 1. Let us assume that the Roman Empire can be described by a graph $G = (V, E)$ as it is presented below, in Figure 1.

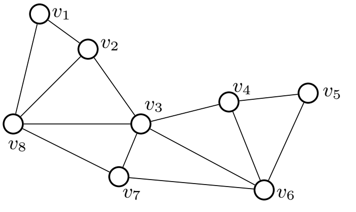

Figure 1: Graph $G = (V, E)$

The optimal number of legions necessary to defend the given graph is 4, provinces represented by vertices v_1 and v_5 are with one stationed legion, province represented by vertex v_3 is with two stationed legions and all other provinces are without stationed legions. With the given schedule, vertices v_1, v_3 and v_5 are defended because they have at least one legion stationed within it, while v_2, v_4, v_6, v_7 and v_8 are defended since they are in the neighborhood of the vertex v_3, which is with two stationed legions. The optimal solution to the proposed problem is illustrated in Figure 2, where vertices are marked by black squares if they are representing provinces with two stationed legions, marked by red circles if they are representing provinces with one stationed legion, and marked by white circles if they are representing provinces without stationed legions.

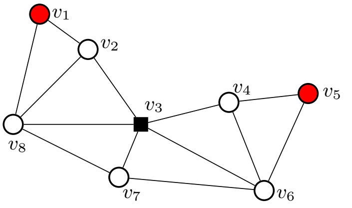

Figure 2: Illustrated solution of the RD problem on a graph G defined in the Example 1

In order to reduce the number of legions necessary to defend the Roman Empire against a single attack, Henning and Hedetniemi [3] introduced the weak Roman domination problem (WRD problem) as a variant of the RD problem. First, they assumed that every province of the Roman Empire is safe if there is at least one legion stationed within it and every unsafe province is defended if it is adjacent to a safe province. Then they required that for every unsafe province there exists at least one adjacent safe province whose legion could move and protect it in case it is attacked, such that this particular legion movement does not affect the Empire's safety, i.e., all provinces are considered to be defended before and after the movement.

Similarly as for the RD problem, for a graph $G = (V, E)$ and a function $f : V \to \{0, 1, 2\}$, every vertex with positive weight is considered to be defended, and a vertex u with property $f(u) = 0$ is considered to be defended if it is adjacent to a vertex $v \in V$ with positive weight. A function f is called a weak Roman dominating function (WRD function) on a graph G if every vertex u with property $f(u) = 0$ is adjacent to a vertex v with property $f(v) > 0$ and, with respect to the function f', $f' : V \to \{0, 1, 2\}$ defined by $f'(u) = 1$, $f'(v) = f(v) - 1$ and $f'(w) = f(w)$, $w \in V \setminus \{u, v\}$, all vertices are defended. The problem of finding the WRD function f with the minimal weight for a given graph G is referred to as the weak Roman domination problem (WRD problem). The minimum weight of the WRD function f, denoted by $\gamma_r(G)$, represents the weak Roman domination number.

We illustrate the weak Roman domination problem in the example below.

Example 2. Let us assume that the Roman Empire can be described by the graph $G = (V, E)$ presented on Figure 1. The optimal solution value for the WRD problem on the given graph is 3. Legions are stationed such that provinces represented by vertices v_1, v_5 and v_7 are with one stationed legion while all other provinces are without stationed legions, see Figure 3 (vertices are marked by red circles if they are representing provinces with one stationed legion and marked by white circles if they are representing provinces without stationed legions).

With the given strategy, in case of an attack, provinces represented by vertices v_2 and v_8 are defended by the legion stationed at the province represented by the vertex v_1. In case of attack, movements of legion stationed at province v_1 to province v_2 or to v_8 does not affect Empire's safety. Similarly, provinces v_4 and v_6 are defended

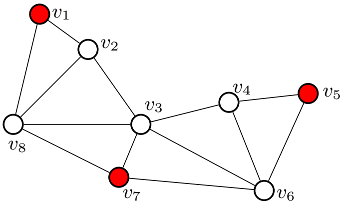

Figure 3: Illustrated solution of the WRD problem for a graph G defined in the Example 2

by the legion from province v_5, province v_3 is defended by the legion from province v_7, etc.

Ivanović [6] showed that neither the CPLEX nor the Gurobi optimization solvers were able to solve the WRD problem on a huge number of instances with more than 100 vertices. Since there is only one algorithm for solving the WRD problem (see [7]), which is written only for block graphs, we present a Variable Neighborhood Search solution for solving the WRD problem on any types of graphs.

We also show that the same algorithm can be applied to the RD problem, although Burger et al. [8] showed that there are significant differences in solving these two problems (their assumption was based on the fact that the RD problem involves static configuration of legions on the vertices of G, while the WRD problem involves moving a legion between the adjacent vertices).

This paper is organized as follows. Previous work is given in Section 2. The Variable Neighborhood Search algorithm is proposed in Section 3. Computational results are summarized in Section 4.

2 PREVIOUS WORK

The Roman domination problem was introduced by Stewart [9] and ReVelle and Rossing [1]. Inspired by Stewart's paper, Cockayne et al. [2] gave some properties of the Roman domination sets. Later Henning et al. [3] introduced the WRD problem as special variant of the RD problem and observed that every RD function in a graph G is also a WRD function in G. In the same paper they proved relation $\gamma(G) \leq \gamma_r(G) \leq \gamma_R(G) \leq 2\gamma(G)$, where $\gamma(G)$ represents cardinality of the minimum dominating set on the graph G (dominating set is a set of vertices such that each of the other vertices has a neighbor in the dominating set). Relations between several different domination numbers were summarized by Chellali et al. [10].

Upper and lower bounds for γ_R for special types of graphs were determined, for instance, in [2, 11, 13, 14, 15, 16, 17]. Exact values for γ_R for paths, cycles, complete, complete n-partite and Petersen $P(n, 2)$ graphs were given in [2, 11, 15, 16, 18, 19, 20, 21, 22], while cardinal and Cartesian products of paths and cycles and lexicographic product of some graphs were given in [15, 16, 19]. Exact values of the $\gamma_r(G)$ for paths, cycles, complete, complete n-partite, $2 \times n$ grid and web graphs and values of $\gamma_r(G)$ of corona and products of some special types of graphs were given in [3, 12, 23, 24].

The complexity of computing γ_R when restricted to interval graphs was mentioned as an open question in [2]. In the same paper it was shown that the problem of computing γ_R on trees can be solved in linear time and that it remains NP-complete even when restricted to split graphs, bipartite graphs, and planar graphs. Linear-time algorithm for computing γ_R on bounded tree-width graphs was proposed in [25]. In [20] it was shown that γ_R can be computed in linear time on interval graphs and co-graphs. In the same papers, the authors give a polynomial time algorithm for computing γ_R on AT-graphs and graphs with d-octopus. Linear-time approximation algorithm and a polynomial time approximation scheme for the RD problem on unit disk graphs was given in [22]. If we assume that the size of G is a given constant, Pavlič and Žerovnik provided algorithm for computing γ_R for polygraphs, including rota-graphs and fascia-graphs, that run in constant time in [19]. Some variants of the algorithm for solving the RD problem on a grid graph together with theoretical properties of γ_R of grid graphs were given in [13]. In [13] Currò also showed that the same algorithm can be applied to some other types of graph.

A binary programing formulations for the RD problem, which can be used for computing γ_R on arbitrary graphs by using standard optimization solvers, were provided by ReVelle and Rossing [1] and Burger et al. [4]. Burger et al. [4] also gave a binary programming formulations for the WRD problem. Recently Ivanović [6] gave another formulation for the WRD problem. Ivanović compared formulations for the WRD problem in [6], showing that neither CPLEX nor Gurobi optimization solvers were able to solve the WRD problem, regardless of the used formulation, on many instances with more than 100 vertices.

Peng [7] gave a linear time algorithm for computing γ_r on block graphs. Providing two faster algorithms, Chapelle et al. [26] broke trivial enumeration barrier of $O^*(3^n)$ for calculating $\gamma_r(G)$ (the notation $O^*(f(n))$ suppresses polynomial factors). With the first algorithm they proved that the WRD problem can be solved in $O^*(2^n)$ time needing exponential space. The second algorithm uses polynomial space and time, $O^*(2.2279^n)$.

For some special classes of graphs (interval graphs, intersection graphs, co-graphs and distance-hereditary graphs) the RD problem can be solved in linear time [15], but in the general case, the RD problem is NP-complete, [11]. Proof that the WRD problem is NP-complete, even when restricted to bipartite and chordal graphs, is given in [3].

Now, since both the Roman and the weak Roman domination problems are NP-complete problems, creating a heuristic that could be successful in finding an optimal solution value, providing legions schedule as well, represents a challenge.

Therefore, in [13] a genetic algorithm for solving the RD problem was proposed by Currò, and that was the only heuristic written for any type of Roman domination problem known to the authors. In the mentioned paper, the author proposes a set of instances on random generated graphs which will be used in experimental results of this paper.

In the next section we propose the Variable Neighborhood Search algorithm for solving both the Roman and the weak Roman domination problems on graphs. The VNS heuristic is chosen because it was previously proven to be successful for some problems on graphs, for example [27, 28].

3 VARIABLE NEIGHBORHOOD SEARCH APPROACH FOR SOLVING ROMAN AND WEAK ROMAN DOMINATION PROBLEMS

The Variable Neighborhood Search (VNS) is a heuristic method, which starts from some point from the search space, explores its neighborhoods, then changes the starting point through some search procedures such that it moves to another point of the search space, explores its neighborhoods, and repeats the whole procedure in order to find a better solution. The VNS heuristic was proposed by Mladenović [29] and later studied by Mladenović and Hansen [30] and Hansen and Mladenović in [31].

With respect to the problems' definitions, let us assume that all Roman provinces are represented by a set of vertices V, $n = |V|$, and all existing roads by the set of edges $E = \{e = (i,j), \quad i,j \in V, \ i \text{ and } j \text{ are connected}\}$, $m = |E|$, of some simple undirected graph $G = (V,E)$. Given that graph G is undirected, we will say that $e = (i,j) \in E$ implies $(j,i) \in E$. Moreover, for every vertex $i \in V$ let the set of all vertices adjacent to the vertex i be marked by N_i. Furthermore, let us assume that each province is represented by a number $i = 1,\ldots,n$, and the number of legions stationed within a province i is represented by value x_i. Vector $X = (x_1,\ldots,x_n)$ of values x_i, $i = 1,\ldots,n$, is a feasible solution to the RD problem (WRD problem) if f, $f : V \to \{0,1,2\}$ defined by

$$f(i) = x_i, \quad i \in V \tag{1}$$

is a Roman domination function (weak Roman domination function).

Given that a feasible solution to the WRD problem does not have to be a feasible solution to the RD problem, we define a function *feasibleSolution*$(X, \textit{problem})$ which checks if X is a feasible solution for the $\textit{problem} \in \{\text{RD, WRD}\}$.

In order to check if vector X is a feasible solution to the RD problem, for every element x_i $(i = 1,\ldots,n)$ *feasibleSolution(X, RD)* checks if x_i is a positive value, or $x_i = 0$ and there is at least one vertex v_j connected to v_i such that $x_j = 2$.

In order to check if vector X is a feasible solution to the WRD problem, for every element x_i $(i = 1,\ldots,n)$ *feasibleSolution(X,WRD)* checks if it is a positive value, or $x_i = 0$ and at least one of the following two conditions holds:

1. there exists at least one element x_j $(j = 1,\ldots,n, \ j \neq i)$ with properties $x_j = 2$ and $j \in N_i$, i.e.

- after a single legion movement from a province j to a province s ($s \neq i, j$) there still is one legion stationed at a province j which defends provinces i and j;
- after a single legion movement from a province j to a province i, both provinces i and j are defended by stationed legions.

2. there exists at least one element x_j, $j \in N_i$, such that $x_j = 1$ and swapping the values of x_i and x_j does not affect the feasibility of the vector X. More precisely, after the swap, for every element x_s, $s \in N_j$, with property $x_s = 0$, there exists at least one x_k, $k \in N_s, k \neq j$, with property $x_k > 0$, i.e.

   - in order to move a single legion from a province j to a province i, all provinces s, which are neighbors with j and which are without any stationed legion, must have another neighbor k ($k \neq j$) with at least one stationed legion.

We will say that the function *feasibleSolution(X, problem)* is satisfied if there are no undefended provinces with respect to the *problem*.

Also, we create function *penalty(X, problem)*, which calculates the number of undefended provinces with respect to the *problem*.

Further, we will say that two solutions, X and X', have difference of the first order if one legion was moved from one province to another (value of one element, with value lower than 2, of the vector X, is increased by one, while value of the other element, with positive value, of the vector X, is decreased by one) or disbanded (value of one element, with positive value, of the vector X, is decreased by one). Respectively, two solutions have difference of the k^{th} order if at most k legions were moved, including possible disbanding.

Now, let us define a set $\mathcal{N}_k(X)$, $k = k_{min}, \ldots, k_{max}$ as the set of all vectors X' that have difference of the k^{th} order from the solution X and call that set k^{th} *Neighborhood to the solution X*.

The VNS-based heuristic can be defined in such a way that it starts from the *initial* feasible solution X, *shakes* it by creating another solution $X' \in \mathcal{N}_k(X)$ (by the expression *shake* we mean *movement of a certain number of legions*) and then applies *local search method* in order to create a better feasible solution X''. If the feasible solution X'', obtained by the local search procedure, is not better than the current incumbent X ($F(X'') \geq F^*$), the VNS algorithm repeats the procedure of shaking, but in neighborhood $\mathcal{N}_{k+k_{step}}(X)$ (i.e., k increments by k_{step}) and local search within it and so on until k reaches its maximum k_{max}. Otherwise, if $F(X'') < F^*$, X^* becomes X'', F^* becomes $F(X'')$ and k becomes k_{min}. Changing neighborhoods enables one to get out from the local minima. The VNS algorithm is presented as Algorithm 1. Functions *InitialSolution()*, *Shake()*, *LocalSearch()* and *StoppingCondition()* are described below.

Function *InitialSolution*() (pseudo code is presented as Algorithm 2) is defined so that it produces an initial feasible solution X^* by applying random changes to elements of the zero vector X. That is, *InitialSolution*() assigns randomly generated number from the set $\{1, 2\}$ to a randomly chosen element of the vector X until X

Algorithm 1 Variable Neighborhood Search metaheuristic

```
X* ← InitialSolution();
F* ← F(X*);
repeat
    k ← k_min;
    repeat
        X ← X*;
        X' ← Shake(X, k);
        X'' ← LocalSearch(X');
        if F(X'') < F* then
            F* ← F(X'');
            X* ← X'';
            k ← k_min;
        else
            k ← k + k_step;
    until k > k_max
until StoppingCondition()
```

Algorithm 2 *InitialSolution*()

```
X ← {0, ..., 0};
repeat
    i ← random number ∈ {1, ..., n};
    x_i ← random number ∈ {1, 2};
until (feasibleSolution(X, problem))
for i = 1, ..., n do
    if x_i > 0 then
        x_i ← x_i − 1;
        if not(feasibleSolution(X, problem)) then
            x_i ← x_i + 1;
```

becomes a feasible solution. Then, given that the function *InitialSolution*() finds a feasible solution, and our goal is to find a feasible solution such that the objective function value $F(X)$ $(F(X) = \sum_{i=1}^{n} x_i)$ is minimal, the found solution will be, for now, saved as the best one $(X^* \leftarrow X,\ F^* \leftarrow F(X^*))$.

Further, in order to lower the value F^*, i.e., to improve the incumbent, among the elements of the vector X with positive value, *InitialSolution*() searches for an element whose value could be decreased by one such that the resulting vector remains a feasible solution. If such an element is found, *InitialSolution*() will decrease its value by one, and then continue to search for an element of the incumbent with the same property. Whenever the procedure of decreasing a value of one element produces a feasible vector, the resulting vector will be stored as the best one and objective function value $F(X)$ will be stored as F^*. This procedure repeats until there are no elements whose decreased value will result with feasible X.

Algorithm 3 *Shake*()

$X \leftarrow X^*$
DecreasingProcedure(X);
for $j = 1, \dots, k$ **do**
    $a \leftarrow$ random number $\in \{1, \dots, n\}$ such that $x_a \neq 0$;
    $b \leftarrow$ random number $\in \{1, \dots, n\}$ such that $x_b \neq 2$;
    $x_a \leftarrow x_a - 1$;
    $x_b \leftarrow x_b + 1$;
if *feasibleSolution*(X, *problem*) **then**
    $X^* \leftarrow X$;
    DecreasingProcedure(X);

Now, if it is possible to find a feasible solution with the same or smaller objective function value than F^*, the resulting solution will be better than the current incumbent. Hence, we define the following two functions, *Shake*() and *LocalSearch*(). These two functions are defined to search for a better feasible solution than the one with which they start the searching process.

Therefore, $Shake(X^*, k)$ function (presented as Algorithm 3) starts with a feasible solution X^*, stores it as X ($X \leftarrow X^*$) and then randomly chooses an element of the solution X with positive value and decreases its value by one. If the resulting vector is again a feasible solution, it stores it as the new best solution and repeats the process until an infeasible solution is found. We call this process *DecreasingProcedure*(). Then, among the elements of the current solution X with value lower than 2, shake function randomly choses one element, and among the elements with positive value of the incumbent X, it randomly chooses another element and increases a value of the first chosen element by one and decreases the value of the second chosen element also by one (i.e., it moves one legion) and repeats this process k times. If the resulting vector X' is a feasible one, given that $F(X') < F^*$ the new best feasible is found. Therefore, X' will be stored as the new best feasible ($X^* \leftarrow X'$). Also, if X' is feasible, we will apply *DecreasingProcedure*() to the vector X' and resulting vector denote as X' (note that in this case it follows that $F(X') \leq F^* - 1$).

Now, the *LocalSearch*(X') function (presented as Algorithms 4 and 5) starts with an infeasible incumbent X', calculates its *penalty*(X', *problem*) value and stores it as nd_{min}. Then it searches a neighborhood $\mathcal{N}_1(X')$ of the incumbent X' in order to find a feasible solution. If a solution with lower penalty value is found it will be stored as incumbent and search for a better solution continues. If a solution with penalty value equal to zero is found, it means that a feasible solution is found. If there is no solution with penalty value lower or equal to nd_{min} within the neighborhood $\mathcal{N}_1(X')$ of the incumbent, local search procedure will continue its search in the neighborhood $\mathcal{N}_2(X')$ of the incumbent. In both cases, whenever a feasible solution is found, it will be stored as the new best feasible solution. Also, local search procedure will continue to search for a feasible solution within the neighborhoods of the incumbent

Algorithm 4 *LocalSearch*()

```
1:  nd_min ← penalty(X', problem);
2:  while some improvement is made do
3:      for i = 1, ..., n  such that x'_i > 0 do
4:          x'_i ← x'_i − 1;
5:          if feasibleSolution(X', problem) then
6:              X* ← X';
7:              DecreasingProcedure(X');
8:              nd_min ← penalty(X', problem);
9:              go to line 3;
10:         else
11:             for j = 1, ..., n, j ≠ i  such that  x'_j < 2  do
12:                 x'_j ← x'_j + 1;
13:                 nd ← penalty(X', problem);
14:                 if nd = 0 then
15:                     execute lines 6-9;
16:                 else
17:                     if nd < nd_min then
18:                         X'_better ← X';
19:                         nd_min ← nd;
20:                     if nd = nd_min then
21:                         X'_same ← X' with some probability;
22:                 x'_j ← x'_j − 1;
23:         x'_i ← x'_i + 1;
24:     if X'_better is found then
25:         X' ← X'_better;
26:     else
27:         if X'_same is found then
28:             with some probability  X' ← X'_same;
29:         else
30:             run LS2();
31: X'' ← X*;
```

(i.e., a decreasing procedure will be applied to the feasible incumbent) until there is no better feasible solution.

In other words, local search procedure consists of three steps. In the first step, local search procedure searches for an element (of the incumbent X') with positive value, decreases its value by one and checks if the resulting vector is a feasible one. If the resulting vector is a feasible solution, it will be stored as X^*. If the resulting vector is infeasible, the procedure goes to the second step of the local search. In the second step, the local search procedure searches for an element $x_j^{'}$ of the incumbent of the local search procedure with property $x_j^{'} < 2$, such that increasing its value by one creates a feasible solution. If the required element is found, its value will be increased by one and the resulting feasible solution stored as X^*. If a feasible solution is found (both in the first and in the second step), $DecreasingProcedure()$ will be applied to that feasible incumbent, nd_{min} will be set to be equal to $penalty(X', problem)$ and the local search procedure will restart from the beginning of the first step (lines 6-9 and 15 of Algorithm 4). If the required element of the second step was not found, solution with the smallest penalty value $penalty(X', problem)$ will be stored as X'_{better} and the solution with the penalty value equal to the incumbent will be stored as X'_{same}. Then, when the second step is finished, in case that a better solution than the incumbent is found, it will be set as the incumbent solution and the second step will restart from the beginning. Similarly, if at least a solution of the same quality is found, it will be set as the incumbent solution with some probability and the second step will restart from the beginning. Otherwise, if there is no better solution nor a solution of the same quality, the third step of the local search procedure will start.

In the third step of the local search procedure, we explore a neighborhood $\mathcal{N}_2(X')$ of the incumbent in order to find a feasible solution. We denoted the third step of the local search procedure as $LS2()$ only because we want to make algorithm of $LocalSearch()$ function easier for reading.

In the third step (which is presented as Algorithm 5), the local search procedure searches for an element x'_i with value $x'_i = 2$ and for an element x'_j with value $x'_j < 2$ $(i, j = 1, \ldots, n)$. Then, it decreases the value of x'_i by two and increases a value of x'_j by one and then checks if a feasible solution is found, or if there exists an element $x'_s < 2$ such that increasing its value by one results with a feasible solution or with a better infeasible solution. Similarly as in the first two steps, $LS2()$ function computes *penalty()* value before and after each change and stores an incumbent solution X' with smaller penalty value than nd_{min} as X'_{better} and the incumbent with the same penalty value as X'_{same}. Again, whenever a better incumbent is found, nd_{min} will be set to be equal to *penalty(*X'_{better}*, problem)* and the incumbent solution of the same quality will be stored with some probability. Then, if a process of decreasing a value of an element x'_i by two and increasing a value of each pair of elements x'_j and x'_s by one does not create a feasible solution, values of elements x_i, x_j and x_s will be restored and the third step will continue its search with the next element whose value is equal to 2. In case that all element combinations are checked and better solution is found, it will be set as the incumbent and $LS2()$ will restart its search within the new incumbent. Similarly, in case that all elements combinations are checked and only a solution of the same quality is found, it will be set as the incumbent with some probability and $LS2()$ will restart.

During all the steps of the local search procedure we are also checking if moves from one solution to the solution of the same quality will not make a loop, i.e., we will not store the incumbent of the same quality if it will take us to some previous incumbent. Given that the size of a loop may vary, we do not allow moves from one incumbent to the incumbent of the same quality for more then k_{max} successive times. This means that the second and the third step will restart with the solution

Algorithm 5 $LS2()$

```
1:  nd_min ← penalty(X', problem)
2:  while some improvement is made do
3:    for i = 1, ..., n such that x'_i = 2 do
4:      x'_i ← x'_i − 2
5:      for j = 1, ..., n such that x'_j < 2 do
6:        x'_j ← x'_j + 1
7:        if feasibleSolution(X', problem) then
8:          X* ← X'
9:          DecreasingProcedure(X')
10:         nd_min ← penalty(X', problem)
11:         go to line 2
12:       else
13:         for s = 1, ..., n, such that x'_s < 2 do
14:           x'_s ← x'_s + 1
15:           if feasibleSolution(X', problem) then
16:             apply lines 8 − 11
17:           else
18:             nd ← penalty(X')
19:             if nd < nd_min then
20:               X'_better ← X'
21:               nd_min ← nd
22:             if nd = nd_min then
23:               X'_same ← X' with some probability
24:           x'_s ← x'_s − 1
25:       x'_j ← x'_j − 1
26:     x'_i ← x'_i + 2
27:   if X'_better is found then
28:     X' ← X'_better
29:   else
30:     if X'_same is found then
31:        with some probability X' ← X'_better
32:     else
33:       finish LS2()
```

of the same quality for no more than k_{max} successive times. If some improvements are made within $LS2()$, the local search procedure restarts from the beginning of the first step with the new incumbent. Finally, when all three steps are finished and no improvement is made, *LocalSearch()* function will finish its search and the feasible solution X^* will be returned as X''. Now, if a better feasible solution is obtained ($F(X'') < F^*$), its objective function value will be stored ($F^* \leftarrow F(X'')$) and k will be set to k_{min}, otherwise k will be increased by k_{step}. The VNS algorithm continues until k reaches its maximum or some other stopping condition occurs.

Input parameters for the VNS heuristic are the *problem*, the minimal (k_{min}) and the maximal (k_{max}) numbers of neighborhoods that should be searched, the increment of the parameter k (k_{step}) and the maximum CPU time allowed (t_{max}). In our implementation *StoppingCondition()* finishes the VNS algorithm if either k_{max} or maximal CPU time allowed is reached.

The parameters used for the proposed VNS algorithm are $k_{min} = 1$, $k_{max} = 30$, $k_{step} = 1$ and $t_{max} = 7\,200$ s and probability is set to $p = 0.5$.

The VNS algorithm cannot guarantee finding global optima because of its non-deterministic nature. Therefore, in order to find solution of sufficiently high quality it is necessary to run the VNS heuristic algorithm on the same instance more than once. Hence, in out experiments each instance was run 20 times.

4 COMPUTATIONAL RESULTS

Experimental results obtained by the proposed VNS algorithm for solving the RD and the WRD problems are presented in this section. The VNS algorithm was implemented in C++. All computational experiments have been performed on Intel® Core™ i7-4700MQ CPU@2.40 GHz with 8 GB RAM, under Windows 10 operating system.

CPLEX optimizations solver was run on all five formulations of the RD problem presented in [5] on grid, planar, net and randomly generated sets of graphs. The set of randomly generated graphs is the same as the one generated and proposed by Currò in [13] (names of instances consist of the number of vertices and of the probability that edge is incident to vertices expressed in percentage) while grid, net and planar sets are well known sets of graphs and also provided by Currò. Since there are several different ILP formulations of the Roman and the weak Roman domination problems (see [5] and [6]), and that performance of CPLEX differs in accordance with used ILP formulation, for the RD problem we present only instances for which optimal solution value is found, while for the WRD problem the results are presented on all instances with some known solution. In case that CPLEX was successful in finding an optimal solution value by using more than one formulation, the smallest running time is presented.

The results are summarized in Tables 1–8.

Tables 1–4 contain instances where CPLEX optimization solver was able to find and prove optimality of the found solution value for the RD problem (CPLEX was run for all five formulations of the RD problem presented in [5]). Tables 5–8 contains instances where CPLEX and Gurobi optimization solvers were successful in finding some solution value by using at least one ILP formulation presented in [6] within the given time. In all tables, whenever the optimal solution value is found by more than one formulation, the smallest running time is shown. Also, whenever optimization solver was unable to prove optimality of the found solution either because of time limit or "out of memory" status, in the column t_{sol} we put sign "–".

Instances are sorted by the number of vertices and the number of edges, in that order. Tables are organized as follows: The name of the instance is given in the first column. The next two columns ($|V|$, $|E|$) represent the number of vertices and the number of edges. In tables that correspond to the RD problem for all instances we have optimal solution values. Therefore, in the next two columns, *opt* and t_{cpl}, optimal solution value and minimal running time are given. In tables that correspond to the WRD problem we have three columns, the optimal solution value, the best solution value and the smallest running time, which is given regardless the optimization solver and ILP formulation. It should be noted that for the WRD problem optimal solution values and minimal running times of standard optimization solvers are taken from [6]. Also note that, in case that optimization solver could not provide an optimal solution value, a symbol "-" stands in the column t_{sol}.

For both problems, the VNS algorithm was run 20 times for each problem instance and informations of the best solution values obtained in these 20 runs are given in the final four columns (sol, t, err, σ) of all the tables. The best solution value obtained by the VNS algorithm is given in the column *sol* and whenever the VNS solution value was equal to the optimal solution value (from *opt* column), it was marked as "*opt*". The best time in 20 runs, necessary for the VNS algorithm to reach the corresponding solution in the first occurrence is given in the column t. The final two columns err and σ contains informations on the average solution quality: err stands for average relative error of found solutions from the best found solution, which is calculated as $err = \frac{1}{20}\sum_{i=1}^{20} err_i$, where $err_i = |\mathrm{VNS}_i - sol|/|\mathrm{VNS}_i|$, and VNS_i is the VNS solution obtained in the i^{th} run. Parameter σ is the standard deviation of the err obtained by the formula $\sigma = \sqrt{\frac{1}{20}\sum_{i=1}^{20}(err_i - err)^2}$.

The VNS algorithm for the RD problem is tested on 231 different instances and achieves the optimal solution on 218 of them. All solutions are found within the time limit (running time for 99 instances is lower than 1 second and only for 29 larger than 100 seconds). For majority of instances (on 214 instances), percentage average relative error from the found solution is lower than 2.5 %. Also, for the majority of instances (for 121 instances) the VNS heuristic running time is lower than the best CPLEX running time. Detailed informations of these testings are given in Tables 1–4.

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	*opt*	t_{cpl}	*sol*	t	err	σ
grid04x10	40	66	20	0.081	opt	0.01	0	0
grid05x08	40	67	21	0.081	opt	0.005	0	0
grid08x05	40	67	21	0.062	opt	< 0.01	0	0
grid10x04	40	66	20	0.077	opt	0.013	0	0
grid03x14	42	67	22	0.042	opt	< 0.01	0	0
grid06x07	42	71	22	0.119	opt	< 0.01	0	0
grid07x06	42	71	22	0.115	opt	< 0.01	0	0
grid14x03	42	67	22	0.062	opt	0.031	0	0

Table 1 continues . . .

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	*opt*	t_{cpl}	*sol*	t	*err*	σ
grid04x11	44	73	22	0.057	opt	0.013	0	0
grid11x04	44	73	22	0.046	opt	< 0.01	0	0
grid03x15	45	72	24	0.046	opt	0.012	0	0
grid05x09	45	76	23	0.168	opt	< 0.01	0	0
grid09x05	45	76	23	0.148	opt	0.013	0	0
grid15x03	45	72	24	0.061	opt	0.022	0	0
grid04x12	48	80	24	0.058	opt	0.034	0	0
grid06x08	48	82	24	0.05	opt	0.033	0.0040	0.0120
grid08x06	48	82	24	0.098	opt	0.012	0.0040	0.0120
grid12x04	48	80	24	0.076	opt	0.019	0	0
grid07x07	49	84	24	0.098	opt	0.029	0.0060	0.0143
grid05x10	50	85	26	0.147	opt	< 0.01	0	0
grid10x05	50	85	26	0.166	opt	< 0.01	0	0
grid04x13	52	87	26	0.162	opt	0.104	0	0
grid13x04	52	87	26	0.099	opt	0.017	0.0019	0.0081
grid06x09	54	93	27	0.111	opt	0.436	0.0143	0.0175
grid09x06	54	93	27	0.179	opt	< 0.01	0.0071	0.0143
grid05x11	55	94	28	0.153	opt	0.013	0	0
grid11x05	55	94	28	0.184	opt	< 0.01	0	0
grid04x14	56	94	28	0.059	opt	0.035	0.0017	0.0075
grid07x08	56	97	28	0.131	opt	< 0.01	0	0
grid08x07	56	97	28	0.153	opt	0.033	0	0
grid14x04	56	94	28	0.06	opt	0.438	0.0356	0.0109
grid04x15	60	101	30	0.092	opt	< 0.01	0.0016	0.0070
grid05x12	60	103	30	0.13	opt	0.036	0.0194	0.0158
grid06x10	60	104	30	0.092	opt	0.041	0.0048	0.0115
grid10x06	60	104	30	0.152	opt	0.163	0.0097	0.0148
grid12x05	60	103	30	0.177	opt	0.078	0.0129	0.0158
grid15x04	60	101	30	0.075	opt	0.04	0	0
grid07x09	63	110	31	0.066	opt	0.135	0.0094	0.0143
grid09x07	63	110	31	0.162	opt	0.082	0	0
grid08x08	64	112	32	0.118	opt	0.031	0.0015	0.0066
grid05x13	65	112	33	0.173	opt	0.171	0.0029	0.0088
grid13x05	65	112	33	0.204	opt	0.054	0.0044	0.0105
grid06x11	66	115	33	0.137	opt	0.045	0.0059	0.0118
grid11x06	66	115	33	0.169	opt	0.246	0.0029	0.0088
grid05x14	70	121	35	0.207	opt	0.264	0.0083	0.0127
grid07x10	70	123	34	0.146	opt	0.164	0.0171	0.0140
grid10x07	70	123	34	0.119	opt	0.699	0.0171	0.0165
grid14x05	70	121	35	0.191	opt	0.19	0.0083	0.0127
grid06x12	72	126	36	0.169	opt	0.198	0.0054	0.0108
grid08x09	72	127	35	0.153	opt	0.017	0.0069	0.0120
grid09x08	72	127	35	0.125	opt	0.037	0.0110	0.0181

Table 1 continues ...

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	*opt*	t_{cpl}	*sol*	t	*err*	σ
grid12x06	72	126	36	0.186	opt	0.161	0.0014	0.0059
grid05x15	75	130	38	0.214	opt	0.352	0.0026	0.0077
grid15x05	75	130	38	0.247	opt	0.101	0	0
grid07x11	77	136	38	0.169	opt	0.094	0.0013	0.0056
grid11x07	77	136	38	0.186	opt	0.102	0.0026	0.0077
grid06x13	78	137	38	0.148	opt	1.553	0.0242	0.0148
grid13x06	78	137	38	0.209	opt	38	0.0256	0.0079
grid08x10	80	142	39	0.128	opt	0.132	0.0113	0.0124
grid10x08	80	142	39	0.142	opt	0.039	0.0075	0.0115
grid09x09	81	144	38	0.073	opt	2.937	0.0226	0.0236
grid06x14	84	148	41	0.134	opt	22.542	0.0306	0.0128
grid07x12	84	149	41	0.168	opt	1.727	0.0083	0.0114
grid12x07	84	149	41	0.192	opt	1.062	0.0024	0.0071
grid14x06	84	148	41	0.231	opt	7.766	0.0225	0.0116
grid08x11	88	157	42	0.192	opt	11.391	0.0275	0.0151
grid11x08	88	157	42	0.141	opt	0.778	0.0206	0.0187
grid06x15	90	159	44	0.247	opt	5.733	0.0188	0.0125
grid09x10	90	161	43	0.223	opt	1.224	0.0279	0.0184
grid10x09	90	161	43	0.237	opt	0.672	0.0102	0.0133
grid15x06	90	159	44	0.264	opt	3.141	0.0133	0.0109
grid07x13	91	162	44	0.178	opt	0.801	0.0177	0.0112
grid13x07	91	162	44	0.178	opt	0.882	0.0177	0.0131
grid08x12	96	172	46	0.21	opt	1.527	0.0178	0.0176
grid12x08	96	172	46	0.191	opt	5.175	0.0159	0.0113
grid07x14	98	175	47	0.247	opt	1.621	0.0247	0.0149
grid14x07	98	175	47	0.214	opt	2.929	0.0196	0.0137
grid09x11	99	178	47	0.194	opt	3.737	0.0124	0.0136
grid11x09	99	178	47	0.287	opt	4.522	0.0245	0.0187
grid10x10	100	180	48	0.22	opt	0.199	0.0051	0.0088
grid08x13	104	187	50	0.262	opt	0.274	0.0097	0.0130
grid13x08	104	187	50	0.401	opt	9.993	0.0146	0.0135
grid07x15	105	188	50	0.348	opt	20.739	0.0252	0.0121
grid15x07	105	188	50	0.278	opt	22.274	0.0243	0.0102
grid09x12	108	195	51	0.268	opt	9.665	0.0244	0.0204
grid12x09	108	195	51	0.29	opt	22.53	0.0208	0.0183
grid10x11	110	199	52	0.306	opt	2.545	0.0185	0.0192
grid11x10	110	199	52	0.256	opt	12.061	0.0222	0.0188
grid08x14	112	202	53	0.289	opt	6.864	0.0201	0.0138
grid14x08	112	202	53	0.284	opt	1.213	0.0228	0.0159
grid09x13	117	212	55	0.232	opt	10.045	0.0260	0.0224
grid13x09	117	212	55	0.439	opt	46.869	0.0262	0.0184
grid08x15	120	217	57	0.404	opt	5.05	0.0154	0.0119
grid10x12	120	218	56	0.236	opt	29.077	0.0381	0.0228

Table 1 continues ...

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	*opt*	t_{cpl}	*sol*	t	*err*	σ
grid12x10	120	218	56	0.326	opt	27.666	0.0343	0.0150
grid15x08	120	217	57	0.414	opt	18.631	0.0161	0.0155
grid11x11	121	220	57	0.443	opt	2.414	0.0219	0.0198
grid09x14	126	229	58	0.25	opt	0.518	0.0397	0.0277
grid14x09	126	229	58	0.334	opt	46.688	0.0458	0.0170
grid10x13	130	237	61	0.519	opt	12.797	0.0174	0.0178
grid13x10	130	237	61	0.529	opt	1.85	0.0188	0.0226
grid11x12	132	241	62	0.453	opt	26.008	0.0248	0.0183
grid12x11	132	241	62	0.464	opt	31.964	0.0204	0.0131
grid09x15	135	246	63	0.535	opt	36.088	0.0252	0.0185
grid15x09	135	246	63	0.733	opt	23.271	0.0312	0.0168
grid10x14	140	256	65	0.478	opt	78.302	0.0339	0.0165
grid14x10	140	256	65	0.432	opt	10.337	0.0359	0.0210
grid11x13	143	262	66	0.463	opt	70.571	0.0303	0.0223
grid13x11	143	262	66	0.503	opt	21.158	0.0372	0.0250
grid12x12	144	264	67	0.516	opt	36.922	0.0349	0.0185
grid10x15	150	275	70	0.715	opt	126.053	0.0266	0.0221
grid15x10	150	275	70	0.951	opt	24.143	0.0301	0.0189
grid11x14	154	283	71	0.483	opt	59.802	0.0438	0.0246
grid14x11	154	283	71	0.67	opt	62.236	0.0382	0.0203
grid12x13	156	287	72	0.715	**73**	115.106	0.0232	0.0159
grid13x12	156	287	72	0.783	opt	62.928	0.0384	0.0168
grid11x15	165	304	76	0.77	opt	117.803	0.0484	0.0198
grid15x11	165	304	76	0.918	opt	52.315	0.0406	0.0193
grid12x14	168	310	77	0.614	opt	181.88	0.0389	0.0205
grid14x12	168	310	77	0.721	opt	155.635	0.0424	0.0222
grid13x13	169	312	78	0.77	opt	68.571	0.0325	0.0191
grid12x15	180	333	82	0.94	**83**	164.384	0.0362	0.0191
grid15x12	180	333	82	1.3	**83**	130.71	0.0439	0.0207
grid13x14	182	337	83	0.777	opt	75.472	0.0486	0.0249
grid14x13	182	337	83	0.776	opt	201.98	0.0441	0.0270
grid13x15	195	362	89	1.73	opt	407.358	0.0483	0.0277
grid15x13	195	362	89	1.309	opt	139.451	0.0460	0.0239
grid14x14	196	364	88	0.739	opt	353.878	0.0516	0.0254
grid14x15	210	391	95	1.198	opt	282.147	0.0508	0.0250
grid15x14	210	391	95	1.159	opt	92.424	0.0543	0.0202
grid15x15	225	420	102	1.357	opt	697.859	0.0536	0.0240
grid20x20	400	760	176	37.579	**185**	676.713	0.0390	0.0135
grid30x20	600	1 150	260	1 279.438	**286**	5 114.624	0.0330	0.0160

Table 1. Experimental results for the RD problem on grid graph instances

From Table 1 it can be concluded that the VNS algorithm reaches the solution value equal to the optimal solution value on almost all instances (unsuccessful on 5 among 133 instances of grid type). On instances "grid12x13", "grid12x15", "grid15x12", "grid20x20" and "grid30x20", where an optimal solution was not reached, percentage average relative error from the found solution is lower than 2.1 %. Further, on 123 of 133 instances, percentage average relative error from the found solution is lower or equal to 2.5 % and on 5 instances between 2.5 % and 3 %. So, from Table 1 we can conclude that for the RD problem on grid graph instances the VNS algorithm provides solutions of good quality and within the time limit.

Instance			CPLEX		VNS			
Name	$\lvert V\rvert$	$\lvert E\rvert$	opt	t_{cpl}	sol	t	err	σ
plan10	10	27	3	0.048	opt	< 0.01	0	0
plan20	20	105	5	0.062	opt	< 0.01	0	0
plan30	30	182	5	0.046	opt	< 0.01	0	0
plan50	50	465	6	0.082	opt	< 0.01	0	0
plan100	100	1 540	10	0.0383	opt	0.054	0	0
plan150	150	2 867	12	1.303	opt	1.166	0	0
plan200	200	4 475	16	145.262	opt	2.466	0	0

Table 2: Experimental results for the RD problem on planar graph instances

From Table 2 it can be concluded that the VNS algorithm reaches the solution value equal to the optimal solution value on all instances with σ equal to zero. The VNS algorithm was also tested on instances "plan250" and "plan300" but, because CPLEX was unable to provide optimal solution values on these instances, we will not present the VNS algorithm results for these instances either. Also, we can conclude that instances of planar type are easier for solving for the VNS algorithm than for CPLEX, given the fact that the VNS algorithm provides results much more rapidly.

Instance			CPLEX		VNS			
Name	$\lvert V\rvert$	$\lvert E\rvert$	opt	t_{cpl}	sol	t	err	σ
Net-10-10	100	342	28	0.043	opt	0.129	0	0
Net-10-20	200	712	56	0.088	opt	18.013	0.0018	0.0053
Net-20-20	400	1 482	98	0.134	opt	944.94	0.0228	0.0316
Net-30-20	600	2 252	140	0.162	**145**	6916.4	0.0580	0.0274

Table 3: Experimental results for the RD problem on net graph instances

From Table 3 it can be concluded that the VNS algorithm reaches the solution value equal to the optimal solution value on 3 of 4 instances. On instance "Net-30-20", where an optimal solution value was not reached, percentage average relative error is equal to 2.74 %. Instances of the net type can be considered as easy for solving for CPLEX given the fact that CPLEX is able to provide results for less than 1 second.

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	opt	t_{cpl}	sol	t	*err*	σ
Random-50-1	50	49	32	0.062	opt	0.031	0	0
Random-50-2	50	49	33	0.062	opt	0.069	0	0
Random-50-3	50	58	28	0.084	opt	0.029	0	0
Random-50-4	50	54	30	0.08	opt	0.006	0	0
Random-50-5	50	67	28	0.1	opt	0.005	0	0
Random-50-6	50	86	25	0.184	opt	0.041	0	0
Random-50-7	50	84	26	0.1	opt	< 0.01	0	0
Random-50-8	50	95	23	0.121	opt	< 0.01	0	0
Random-50-9	50	108	23	0.152	opt	0.011	0	0
Random-50-10	50	112	22	0.162	opt	0.021	0	0
Random-50-20	50	248	12	0.337	opt	< 0.01	0	0
Random-50-30	50	373	9	0.178	opt	< 0.01	0	0
Random-50-40	50	475	8	0.432	opt	< 0.01	0	0
Random-50-50	50	597	6	0.285	opt	< 0.01	0	0
Random-50-60	50	739	4	0.115	opt	< 0.01	0	0
Random-50-70	50	860	4	0.121	opt	< 0.01	0	0
Random-50-80	50	980	4	0.131	opt	< 0.01	0	0
Random-50-90	50	1 103	3	0.131	opt	< 0.01	0	0
Random-100-1	100	100	61	0.062	opt	4.662	0.0056	0.0092
Random-100-2	100	109	59	0.1	opt	2.744	0.0058	0.0095
Random-100-3	100	181	48	0.168	opt	3.767	0.0142	0.0113
Random-100-4	100	206	45	0.438	opt	0.895	0.0184	0.0103
Random-100-5	100	231	39	0.469	opt	3.425	0.0243	0.0251
Random-100-6	100	321	34	0.532	opt	3.572	0.0157	0.0142
Random-100-7	100	317	32	0.585	opt	3.291	0.0152	0.0152
Random-100-8	100	398	29	0.774	opt	0.669	0.0017	0.0073
Random-100-9	100	430	27	0.728	opt	0.389	0	0
Random-100-10	100	498	24	1.263	opt	3.95	0.0160	0.0196
Random-100-20	100	981	14	0.971	opt	0.086	0	0
Random-100-30	100	1 477	11	2.916	opt	0.137	0.0083	0.0250
Random-100-40	100	1 945	8	0.761	opt	0.052	0	0
Random-100-50	100	2 483	7	0.808	opt	0.049	0.0188	0.0446
Random-100-60	100	2 985	6	0.345	opt	< 0.01	0	0
Random-100-70	100	3 435	5	0.285	opt	0.044	0	0
Random-100-80	100	3 935	4	0.238	opt	< 0.01	0	0
Random-100-90	100	4 446	4	0.263	opt	< 0.01	0	0
Random-150-1	150	157	94	0.115	opt	22.389	0.0011	0.0032
Random-150-2	150	243	78	0.332	opt	234.872	0.0290	0.0151
Random-150-3	150	322	65	0.834	opt	67.784	0.0171	0.0162
Random-150-4	150	437	53	1.046	opt	30.304	0.0264	0.0155
Random-150-5	150	557	46	3.115	opt	2.293	0.0169	0.0142
Random-150-6	150	705	38	10.362	opt	19.279	0.0165	0.0165

Table 4 continues ...

Instance			CPLEX		VNS			
Name	$\|V\|$	$\|E\|$	opt	t_{cpl}	sol	t	*err*	σ
Random-150-7	150	778	34	5.622	opt	0.462	0.0057	0.0114
Random-150-8	150	906	31	18.691	opt	0.865	0	0
Random-150-9	150	965	30	10.489	opt	3.727	0.0064	0.0161
Random-150-10	150	1 152	27	45.44	opt	3.128	0.0054	0.0128
Random-150-20	150	2 228	16	31.857	opt	1.561	0	0
Random-150-30	150	3 318	12	21.507	opt	0.383	0	0
Random-150-40	150	4 476	9	13.628	opt	0.409	0.0700	0.0458
Random-150-50	150	5 550	8	17.671	opt	0.014	0	0
Random-150-60	150	6 734	6	1.742	opt	0.012	0	0
Random-150-70	150	7 807	6	8.667	opt	0.015	0	0
Random-150-80	150	8 924	4	0.366	opt	0.019	0	0
Random-150-90	150	10 043	4	0.839	opt	0.017	0	0
Random-200-1	200	229	116	0.132	**117**	173.552	0.0167	0.0119
Random-200-2	200	390	92	0.933	**93**	647.247	0.0294	0.0184
Random-200-3	200	581	69	2.69	opt	507.393	0.0403	0.0256
Random-200-4	200	737	60	13.301	opt	568.08	0.0433	0.0214
Random-200-5	200	1 010	47	60.589	opt	41.339	0.0354	0.0217
Random-200-6	200	1 180	42	245.778	opt	84.363	0.0518	0.0332
Random-200-7	200	1 453	36	130.93	opt	11.272	0.0093	0.0173
Random-200-30	200	5 876	12	153.586	opt	9.478	0.0110	0.0346
Random-200-40	200	7 907	10	89.663	opt	0.302	0	0
Random-200-50	200	9 895	8	30.844	opt	0.248	0	0
Random-200-60	200	11 971	6	7.707	opt	0.496	0	0
Random-200-70	200	14 059	6	19.27	opt	0.025	0	0
Random-200-80	200	15 918	4	0.831	opt	0.038	0	0
Random-200-90	200	17 821	4	0.801	opt	0.03	0	0
Random-250-1	250	345	136	0.21	**137**	1 111.594	0.0220	0.0130
Random-250-2	250	633	97	7.95	**99**	380.006	0.0304	0.0211
Random-250-3	250	956	73	257.891	opt	132.791	0.0305	0.0252
Random-250-4	250	1 194	62	1 406.04	opt	148.167	0.0224	0.0218
Random-250-30	250	9 347	13	1 408.412	**14**	1.005	0	0
Random-250-40	250	12 500	10	359.601	opt	0.743	0	0
Random-250-50	250	15 605	8	61.927	opt	0.621	0	0
Random-250-60	250	18 660	8	206.548	opt	0.037	0	0
Random-250-70	250	21 741	6	40.379	opt	0.037	0	0
Random-250-80	250	24 836	4	3.071	opt	0.465	0	0
Random-250-90	250	27 974	4	1.404	opt	0.052	0	0
Random-300-1	300	481	145	0.299	**149**	2 797.158	0.0221	0.0135
Random-300-2	300	876	103	116.818	**105**	1 057.238	0.0394	0.0192
Random-300-40	300	17 934	10	483.378	opt	3.232	0.0174	0.0437
Random-300-50	300	22 520	8	334.329	opt	31.909	0	0
Random-300-60	300	26 952	8	622.751	opt	0.069	0	0
Random-300-70	300	31 390	6	66.546	opt	0.286	0	0

Table 4 continues ...

Instance			CPLEX		VNS			
Name	$\lvert V\rvert$	$\lvert E\rvert$	opt	t_{cpl}	sol	t	*err*	σ
Random-300-80	300	35 871	5	34.579	opt	1.725	0.0667	0.0816
Random-300-90	300	40 412	4	2.191	opt	0.092	0	0

Table 4. Experimental results for the RD problem on random graph instances

Table 4 contains the results of the experimental testing on random generated graphs. As it can be seen, the VNS algorithm reaches the solution value equal to the optimal solution value on many instances (unsuccessful on 7 among 87 instances). On instances where an optimal solution was not reached, standard deviation σ is lower than 2.5 %. Instances "Random-200-8"–"Random-200-20", "Random-250-5"–"Random-250-20" and "Random-300-3"–"Random-300-30" are omitted from Table 4 because CPLEX was unable to find an optimal solution value on these instances. Nevertheless, the VNS algorithm finds some solution value for these instances, but because we do not have an optimal solution value on these instances, we will not present the VNS algorithm results either.

Before we present experimental results for the WRD problem on the same set of instances, let us summarize the results presented in Tables 1-4. The VNS algorithm for the RD problem finds solutions of good quality relatively fast, especially on instances of planar type. On instances of grid and net type, using CPLEX optimization solver is better, but on instances of planar and random type, using the VNS algorithm is preferable.

Experimental results of the VNS algorithm for the WRD problem are performed on instances where some solution values are known from the literature. Given that CPLEX was not able to solve the WRD problem on many instances within the time limit because of the "out of memory" status or because of the time limit, we tested the VNS algorithm both on instances where the optimal solution value is known and on instances where the found solution is not proved to be the optimal solution. Testings were made on 84 instances of different type. CPLEX optimization solver was able to find the optimal solution on 64 of them. The VNS algorithm was not able to find solutions equal to the optimal ones only on two instances. On instances where the optimal solution value is unknown, the VNS solutions are equal or better than the solutions found by CPLEX. Also, for almost all instances, the VNS algorithm runtime is lower than CPLEX runtime. Detailed information considering these testings is provided in Tables 5–8.

From Table 5 it can be concluded that the VNS reaches the solution value equal to the optimal solution value on almost all instances (unsuccessful only on "grid06x13"). On instances where the optimal solution value is unknown, σ is lower than 2.2 %. Running times on instances where the optimal solution value is known shows that the VNS rapidly reaches these solutions in lower than 150 seconds. Even more, on many instances (38 of 42), running times are smaller than 30 seconds and only on "grid07x14" and "grid08x12" greater than 100 seconds. On instances where

Instance				Solver		VNS			
Name	$\|V\|$	$\|E\|$	opt	val	t	sol	t	*err*	σ
grid04x10	40	66	15	15	4.109	opt	0.015	0	0
grid05x08	40	67	14	14	4.64	opt	0.047	0.0333	0.0333
grid03x14	42	67	16	16	4.829	opt	< 0.01	0	0
grid06x07	42	71	15	15	5.801	opt	0.08	0.0063	0.0188
grid04x11	44	73	16	16	5.5	opt	0.031	0.0088	0.0210
grid03x15	45	72	17	17	7.789	opt	0.012	0	0
grid05x09	45	76	16	16	7.908	opt	0.139	0.0235	0.0288
grid04x12	48	80	17	17	12.84	opt	0.069	0.0361	0.0265
grid06x08	48	82	18	18	25.499	opt	< 0.01	0	0
grid07x07	49	84	18	18	9.845	opt	0.021	0	0
grid05x10	50	85	18	18	10.61	opt	0.055	0.0053	0.0158
grid04x13	52	87	19	19	11.813	opt	0.035	0.0050	0.0150
grid06x09	54	93	19	19	25.539	opt	0.331	0.0450	0.0150
grid05x11	55	94	19	19	11.424	opt	0.388	0.0300	0.0245
grid04x14	56	94	20	20	35.326	opt	0.082	0.0214	0.0237
grid07x08	56	97	20	20	21.882	opt	0.076	0.0286	0.0233
grid04x15	60	101	22	22	40.256	opt	0.163	0	0
grid05x12	60	103	21	21	14.88	opt	4.036	0.0271	0.0260
grid06x10	60	104	21	21	35.713	opt	0.746	0.0273	0.0223
grid07x09	63	110	22	22	70.259	opt	0.318	0.0370	0.0155
grid08x08	64	112	23	23	171.925	opt	0.037	0.0063	0.0149
grid05x13	65	112	23	23	67.007	opt	0.928	0.0208	0.0208
grid06x11	66	115	24	24	381.771	opt	0.757	0.0040	0.0120
grid05x14	70	121	24	24	73.489	opt	27.03	0.0491	0.0202
grid07x10	70	123	25	25	618.089	opt	0.67	0.0077	0.0154
grid06x12	72	126	26	26	1 166.405	opt	0.544	0.0074	0.0148
grid08x09	72	127	25	25	435.146	opt	15.935	0.0383	0.0117
grid05x15	75	130	26	26	288.06	opt	8.133	0.0313	0.0174
grid07x11	77	136	27	27	988.596	opt	0.582	0.0268	0.0155
grid06x13	78	137	27	27	1 005.126	**28**	0.407	0.0086	0.0149
grid08x10	80	142	28	28	2 162.812	opt	10.011	0.0375	0.0178
grid09x09	81	144	28	28	737.579	opt	12.521	0.0437	0.0251
grid06x14	84	148	30	30	–	30	2.319	0.0097	0.0148
grid07x12	84	149	29	29	4 637.38	opt	47.642	0.0441	0.0181
grid08x11	88	157	31	31	–	31	3.412	0.0278	0.0190
grid06x15	90	159	32	32	–	32	1.196	0.0179	0.0218
grid09x10	90	161	31	31	–	31	40.651	0.0443	0.0197
grid07x13	91	162	32	32	–	32	16.778	0.0272	0.0130
grid08x12	96	172	33	33	–	33	107.765	0.0403	0.0184
grid07x14	98	175	34	34	1 720.86	opt	143.804	0.0433	0.0181
grid09x11	99	178	35	35	–	35	2.261	0.0181	0.0132
grid10x10	100	180	35	35	–	35	6.63	0.0302	0.0168

Table 5: Experimental results for the WRD problem on grid graph instances

optimization solvers were unable to prove optimality of the found solutions, the VNS heuristic reaches the same solution values for less than 108 seconds. So, we can conclude that the VNS heuristic solves the WRD problem on grid graph instance significantly faster than the optimization solver CPLEX and found solutions are of good quality.

Instance				Solver		VNS			
Name	$\|V\|$	$\|E\|$	opt	val	t	sol	t	*err*	σ
plan10	10	27	3	3	0.156	opt	< 0.01	0	0
plan20	20	105	3	3	1.36	opt	< 0.01	0	0
plan30	30	182	5	5	7.49	opt	< 0.01	0	0
plan50	50	465	6	6	98.49	opt	0.01	0	0
plan100	100	1 540	9	9	–	<u>8</u>	4.916	0	0
plan150	150	2 867	13	13	–	<u>10</u>	88.248	0.0273	0.041

Table 6: Experimental results for the WRD problem on planar graph instances

From Table 6 it can be concluded that the VNS algorithm reaches the solution value equal to the optimal solution value on all instances. Also, on instances where optimization solvers were unable to prove optimality of the found solution, the VNS solution is better. Again, running time for the instances where the optimal solution value is known is lower than 1 second. On "plan100", where optimization solvers were unable to prove optimality of the found solution, the proposed VNS algorithm finds solution value with σ equal to zero. On "plan150" the VNS solution is equal to 10 with $\sigma = 0.0417$, which can be considered as the solution of the good quality (solution value equal to 10 was reached in 14 of 20 runnings).

Instance				Solver		VNS			
Name	$\|V\|$	$\|E\|$	opt	val	t	sol	t	*err*	σ
Net-10-10	100	342	20	20	148.213	opt	4.29	0.0095	0.0190
Net-10-20	200	712	40	40	–	40	67.323	0.0146	0.0119
Net-20-20	400	1 482	83	83	–	<u>81</u>	2 066.577	0.0180	0.0132
Net-30-20	600	2 252	122	122	–	**123**	6 034.018	0.0474	0.0352

Table 7: Experimental results for the WRD problem on net graph instances

In Table 7 optimization solvers were able to find optimal solution value only for "Net-10-10". The same solution value was found by the proposed VNS algorithm with lower running time and with σ equal to 1.9 %. On "Net-10-20" and "Net-20-20" the VNS algorithm reaches the same and better solution value than optimization solvers, while for "Net-30-20" the VNS solution value is worse than the solvers' solution value.

From Table 8 it can be concluded that the VNS algorithm reaches the solution value equal to the optimal solution value on almost all instances (unsuccessful only on 1 among 25 instances of random type). On instance "Random-100-6", where the

Instance				Solver		VNS			
Name	$\|V\|$	$\|E\|$	opt	val	t	sol	t	err	σ
Random-50-1	50	49	24	24	0.281	opt	< 0.01	0	0
Random-50-2	50	49	23	23	0.343	opt	0.034	0	0
Random-50-3	50	58	24	24	0.39	opt	0.062	0	0
Random-50-4	50	54	24	24	0.484	opt	0.225	0	0
Random-50-5	50	67	22	22	0.968	opt	0.377	0.0196	0.0216
Random-50-6	50	86	19	19	2.053	opt	0.03	0	0
Random-50-7	50	84	19	19	3.171	opt	0.889	0.0175	0.0238
Random-50-8	50	95	17	17	3.093	opt	0.131	0.0333	0.0272
Random-50-9	50	108	17	17	26.373	opt	0.129	0.0028	0.0121
Random-50-10	50	112	16	16	6.781	opt	0.047	0	0
Random-50-20	50	248	9	9	346.264	opt	< 0.01	0	0
Random-50-30	50	373	7	7	476.278	opt	0.038	0	0
Random-50-40	50	475	6	6	1447.318	opt	0.092	0	0
Random-50-50	50	597	5	5	1545.06	opt	0.013	0	0
Random-50-60	50	739	4	4	210.71	opt	0.014	0	0
Random-50-70	50	860	3	3	156.14	opt	0.059	0	0
Random-50-80	50	980	3	3	90.813	opt	< 0.01	0	0
Random-50-90	50	1103	2	2	36.53	opt	0.03	0	0
Random-100-1	100	100	46	46	0.64	opt	157.329	0.0354	0.0145
Random-100-2	100	109	46	46	0.843	opt	36.052	0.0148	0.0117
Random-100-3	100	181	37	37	7.421	opt	23.64	0.0445	0.0261
Random-100-4	100	206	34	34	61.702	opt	12.367	0.0213	0.0175
Random-100-5	100	231	32	32	164.502	opt	60.361	0.0299	0.0186
Random-100-6	100	321	26	26	5 806.74	**27**	12.441	0.0265	0.0217
Random-100-7	100	317	25	25	4 009.377	opt	204.939	0.0434	0.0234
Random-100-8	100	317	23	23	–	23	313.924	0.0448	0.0279
Random-100-9	100	430	21	21	–	21	4.98	0.0269	0.0293
Random-100-10	100	498	19	19	–	19	460.905	0.0445	0.0260
Random-100-20	100	981	12	12	–	11	8.951	0.0250	0.0382
Random-100-30	100	1 477	11	11	–	8	1 462.462	0.1056	0.0242
Random-100-40	100	1 945	9	9	–	7	1.501	0	0
Random-100-50	100	2 483	7	7	–	5	37.134	0	0

Table 8: Experimental results for the WRD problem on random generated graph instances

optimal solution value was not reached, σ is equal to 2.17 %. Further, on instances "Random-100-40" and "Random-100-50", where optimization solvers were unable to prove optimality of the found solution, the VNS algorithm finds better solutions values with σ equal to zero for less than 38 seconds.

From Tables 5–8 we can see that optimization solvers were unable to provide an optimal solution value on instances of grid type with number of vertices larger than 84, on instances of planar and net type with number of vertices larger than

100 and on large number of instances of random type with 100 vertices. Also, we can see that, on the same set of instances, the VNS algorithm finds solutions of the WRD problem of good quality and, for many instances, faster than optimization solvers.

5 CONCLUSIONS

In this paper, the Variable Neighborhood Search approach for solving the Roman and the weak Roman domination problems is proposed. Tests were run on grid, net, planar and randomly generated graphs, with up to 600 vertices. The VNS was able to find solutions equal to the optimal ones for the RD problem on 218 of 231 tested instances and able to find solutions equal or better than CPLEX solutions for the WRD problem on 84 of 86 tested instances. Therefore, we can conclude that the VNS algorithm provides good quality solutions regardless of the type of instance and the type of problem, which makes it efficient for solving both the Roman and the weak Roman domination problems. Moreover, given the fact that optimization solvers were not able to solve the WRD problem on large scale instances (i.e., instances with more than 100 vertices) proposed algorithm can be used. Furthermore, given the fact that this algorithm does not contain any limitations on the number of variables and the number of conditions, it can be used for solving the RD problem on instances where optimization solvers are not able to provide an optimal solution value.

In future work, hybridization with some exact methods or application of some other heuristic could lead to possible better achievements in solving the Roman and the weak Roman domination problems.

Acknowledgments

This research has been partially supported by the Serbian Ministry of Education, Science and Technological Development under grants No. TR36006 and ON174010. The authors gratefully acknowledge the two referees of this paper for many excellent suggestions which have helped in improving this paper.

REFERENCES

[1] ReVelle, C. S.—Rosing, K. E.: Defendens Imperium Romanum: A Classical Problem in Military Strategy. The American Mathematical Monthly, Vol. 107, 2000, No. 7, pp. 585–594, doi: 10.2307/2589113.

[2] Cockayne, E. J.—Dreyer, P. A.—Hedetniemi, S. M.—Hedetniemi, S. T.: Roman Domination in Graphs. Discrete Mathematics, Vol. 278, 2004, No. 1-3, pp. 11–22, doi: 10.1016/j.disc.2003.06.004.

[3] Henning, M. A.—Hedetniemi, S. T.: Defending the Roman Empire – A New Strategy. Discrete Mathematics, Vol. 266, 2003, No. 1-3, pp. 239–251, doi: 10.1016/s0012-365x(02)00811-7.

[4] Burger, A. P.—de Villiers, A. P.—van Vuuren, J. H.: A Binary Programming Approach Towards Achieving Effective Graph Protection. Proceedings of the 2013 ORSSA Annual Conference, ORSSA, 2013, pp. 19–30.

[5] Ivanović, M.: Improved Mixed Integer Linear Programing Formulations for Roman Domination Problem. Publications de l'Institut Mathématique, Vol. 99, 2016, No. 113, pp. 51–58, doi: 10.2298/PIM1613051I.

[6] Ivanović, M.: Improved Integer Linear Programming Formulation for Weak Roman Domination Problem. Soft Computing, Vol. 22, 2018, No. 19, pp. 6583–6593, doi: 10.1007/s00500-017-2706-4.

[7] Liu, C. S.—Peng, S. L.—Tang, C. Y.: Weak Roman Domination on Block Graphs. Proceedings of the 27th Workshop on Combinatorial Mathematics and Computation Theory, Providence University, Taichung, Taiwan, April 30–May 1, 2010, pp. 86–89.

[8] Burger, A. P.—Cockayne, E. J.—Grundlingh, W. R.—Mynhardt, C. M.—Van Vuuren, J. H.—Winterbach, W.: Finite Order Domination in Graphs. Journal of Combinatorial Mathematics and Combinatorial Computing, Vol. 49, 2004, pp. 159–176.

[9] Stewart, I.: Defend the Roman Empire! Scientific American, Vol. 281, 1999, pp. 136–138, doi: 10.1038/scientificamerican1299-136.

[10] Chellali, M.—Haynes, T. W.—Hedetniemi, S. T.: Bounds on Weak Roman and 2-Rainbow Domination Numbers. Discrete Applied Mathematics, Vol. 178, 2014, pp. 27–32, doi: 10.1016/j.dam.2014.06.016.

[11] Dreyer Jr, P. A.: Applications and Variations of Domination in Graphs. Ph.D. thesis, Rutgers University, 2000.

[12] Cockayne, E. J.—Grobler, P. J. P.—Gründlingh, W. R.—Munganga, J.—van Vuuren, J. H.: Protection of a Graph. Utilitas Mathematica, Vol. 67, 2005, pp. 19–32.

[13] Currò, V.: The Roman Domination Problem on Grid Graphs. Ph.D. thesis, Università di Catania, 2014.

[14] Favaron, O.—Karami, H.—Khoeilar, R.—Sheikholeslami, S. M.: On the Roman Domination Number of a Graph. Discrete Mathematics, Vol. 309, 2009, No. 10, pp. 3447–3451, doi: 10.1016/j.disc.2008.09.043.

[15] Klobučar, A.—Puljić, I.: Some Results for Roman Domination Number on Cardinal Product of Paths and Cycles. Kragujevac Journal of Mathematics, Vol. 38, 2014, No. 1, pp. 83–94, doi: 10.5937/KgJMath1401083K.

[16] Klobučar, A.—Puljić, I.: Roman Domination Number on Cardinal Product of Paths and Cycles. Croatian Operational Research Review, Vol. 6, 2015, No. 1, pp. 71–78, doi: 10.17535/crorr.2015.0006.

[17] Xing, H.-M.—Chen, X.—Chen, X.-G.: A Note on Roman Domination in Graphs. Discrete Mathematics, Vol. 306, 2006, No. 24, pp. 3338–3340, doi: 10.1016/j.disc.2006.06.018.

[18] Wang, H.—Xu, X.—Yang, Y.—Ji, C.: Roman Domination Number of Generalized Petersen Graphs $p(n, 2)$. arXiv Preprint, arXiv:1103.2419, 2011.

[19] Pavlič, P.—Žerovnik, J.: Roman Domination Number of the Cartesian Products of Paths and Cycles. The Electronic Journal of Combinatorics, Vol. 19, 2012, No. 3, Art. No. P19.

[20] Liedloff, M.—Kloks, T.—Liu, J.—Peng, S.-L.: Efficient Algorithms for Roman Domination on Some Classes of Graphs. Discrete Applied Mathematics, Vol. 156, 2008, No. 18, pp. 3400–3415, doi: 10.1016/j.dam.2008.01.011.

[21] Liedloff, M.—Kloks, T.—Liu, J.—Peng, S. L.: Roman Domination over Some Graph Classes. In: Kratsch, D. (Ed.): Graph-Theoretic Concepts in Computer Science (WG 2005). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 3787, 2005, pp. 103–114, doi: 10.1007/11604686_10.

[22] Shang, W.—Hu, X.: The Roman Domination Problem in Unit Disk Graphs. In: Shi, Y., van Albada, G. D., Dongarra, J., Sloot, P. M. A. (Eds.): Computational Science (ICCS 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4489, 2007, pp. 305–312, doi: 10.1007/978-3-540-72588-6_51.

[23] Pushpam, P. R. L.—Malini Mai, T. N. M.: Weak Roman Domination in Graphs. Discussiones Mathematicae Graph Theory, Vol. 31, 2011, No. 1, pp. 161–170, doi: 10.7151/dmgt.1532.

[24] Lai, Y. L.—Lin, C. T.—Ho, H. M.: Weak Roman Domination on Graphs. Proceedings of the 28$^{\text{th}}$ Workshop on Combinatorial Mathematics and Computation Theory, National Penghu University of Science and Technology, Penghu, Taiwan, May 27–28, 2011, pp. 224–214.

[25] Peng, S.-L.—Tsai, Y.-H.: Roman Domination on Graphs of Bounded Treewidth. Proceedings of the 24$^{\text{th}}$ Workshop on Combinatorial Mathematics and Computation Theory, 2007, pp. 128–131.

[26] Chapelle, M.—Cochefert, M.—Couturier, J.-F.—Kratsch, D.—Liedloff, M.—Perez, A.: Exact Algorithms for Weak Roman Domination. In: Lecroq, T., Mouchard, L. (Eds.): Combinatorial Algorithms (IWOCA 2013). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 8288, 2013, pp. 81–93, doi: 10.1007/978-3-642-45278-9_8.

[27] Hansen, P.—Mladenović, N.—Urošević, D.: Variable Neighborhood Search for the Maximum Clique. Discrete Applied Mathematics, Vol. 145, 2004, No. 1, pp. 117–125, doi: 10.1016/j.dam.2003.09.012.

[28] Brimberg, J.—Mladenović, N.—Urošević, D.—Ngai, E.: Variable Neighborhood Search for the Heaviest k-Subgraph. Computers and Operations Research, Vol. 36, 2009, No. 11, pp. 2885–2891, doi: 10.1016/j.cor.2008.12.020.

[29] Mladenović, N.: A Variable Neighborhood Algorithm – A New Metaheuristic for Combinatorial Optimization. Papers Presented at Optimization Days, 1995, p. 112.

[30] Mladenović, N.—Hansen, P.: Variable Neighborhood Search. Computers and Operations Research, Vol. 24, 1997, No. 11, pp. 1097–1100, doi: 10.1016/s0305-0548(97)00031-2.

[31] Hansen, P.—Mladenović, N.: An Introduction to Variable Neighborhood Search. In: Voss, S., Martello, S., Osman, I. H., Roucairol, C. (Eds.): Meta-Heuristics. Springer, Boston, MA, 1999, pp. 433–458, doi: 10.1007/978-1-4615-5775-3_30.

Marija Ivanović finished her master studies at Faculty of Mathematics, University of Belgrade, in 2011. Since 2007 she has worked at the Faculty of Mathematics, Department for Numerical Mathematics and Optimization as Assistant. The main areas of research are game theory, combinatorial optimization and operations research. She participates in research projects financed by the Ministry of Education, Science and Technological Development, Serbia.

Dragan Urošević finished his Ph.D. studies at Faculty of Mathematics, University of Belgrade, in 2004. Since 1993 he has worked at the Mathematical Institute SANU. The main areas of research are combinatorial optimization and operations research. He is engaged in the development and implementation of heuristic methods to solve complex problems in graph theory and the development of methods for solving location problems. He participates in research projects financed by the Ministry of Education, Science and Technological Development, Serbia.