A NEW VARIABLE NEIGHBORHOOD SEARCH APPROACH FOR SOLVING DYNAMIC MEMORY ALLOCATION PROBLEM

Marija IVANOVIĆ
Faculty of Mathematics, University of Belgrade, Serbia
marijai@math.rs

Aleksandar SAVIĆ
Faculty of Mathematics, University of Belgrade, Serbia
asavic@matf.bg.ac.rs

Dragan UROŠEVIĆ
Mathematical Institute, SANU, Belgrade, Serbia
draganu@mi.sanu.ac.rs

Djordje DUGOŠIJA
Faculty of Mathematics, University of Belgrade, Serbia
dugosija@matf.bg.ac.rs

Received: October 2016 / Accepted: May 2018

Abstract: This paper is devoted to the Dynamic Memory Allocation Problem (DMAP) in embedded systems. The existing Integer Linear Programing (ILP) formulation for DMAP is improved, and given that there are several metaheuristic approaches for solving the DMAP, a new metaheuristic approach is proposed and compared with the former ones. Computational results show that our new heuristic approach outperforms the best algorithm found in the literature regarding quality and running times.

Keywords: Dynamic Memory Allocation Problem, Combinatorial Optimization, Metaheuristics, Variable Neighborhood Search.

MSC: 90C59, 05C90, 68T20.
1. INTRODUCTION

Dedicated computational systems within a larger mechanical or electrical system, usually with real time constraints, or the embedded systems, represent an integral part of a large number of devices, ranging from portable devices up to large industrial, medical, and military structures. Minimizing energy consumption while increasing reliability and performance present a major challenge for engineers. Therefore, designers want to find a balance between the architecture cost and its power consumption [1].

Power consumption of a given application can be reduced by using data access parallelization, which leads us to the definition of the Dynamic Memory Allocation Problem (DMAP).

DMAP can be described as follows. If an application involves a number of data structures with the given size and access cost, a number of memory banks with limited memory capacity, and one external memory bank with unlimited capacity, develop the best memory allocation scheme so that change of data structure allocation during program execution is allowed, while keeping loading, moving, and access time for those and all other operations to a minimum.

Given that modern devices are usually designed to run on processors with integrated or external memory, this problem has become very popular.

DMAP, as defined above, has an important role in modern software since the dynamic memory usage provides greater flexibility and functionality of the applications. There are a large number of papers and references to this particular issue in the literature. In this paper, DMAP in embedded systems is studied as an execution time problem, shown by Wuytack et al. in [17].

A memory allocation problem in embedded systems was widely analyzed by Soto et al. in [14]. In [12] Soto et al. proposed the first Mixed Integer Linear Programing (MILP) formulation for the static version of the memory allocation problem and a metaheuristic approach based on the Variable Neighborhood Search (VNS). Later, Soto et al. [13] dealt with a dynamic version of the memory allocation problem, providing ILP formulation, and two iterative approaches for solving DMAP in embedded systems. These two approaches were followed by GRASP with the ejection chains for the memory allocation problem in embedded systems proposed in Sevaux et al. [11]. Focused on improving the memory allocation problem in embedded systems, Sánchez-Oro et al. [15] and [16] recently proposed a parallel VNS algorithm for DMAP and compared it with the previous ones. More precisely, they focused on the Synchronous Parallel VNS (SPVNS) variant that was used for parallelization of the local search method in the sequential VNS (in [15]) and on the Replicated Shaking VNS (RSVNS), which allows the search to simultaneously explore more solutions in the current neighborhood (in [16]).

In this paper, an improved ILP formulation for DMAP in embedded systems is proposed and a new metaheuristic approach based on the VNS is given. Results of this new metaheuristic approach are compared with the results from the existing literature.
The paper is organized as follows: Problem notation and ILP formulation known from the literature are given in Section 2. Section 3 is dedicated to the improved ILP formulation for dynamic versions of memory allocation problem followed by a short illustrative example. A new VNS approach is presented in the next section. Finally, the existing results from the literature, and the results achieved by using new VNS approach are given and compared in Section 5.

2. EXISTING MATHEMATICAL FORMULATION AND PROBLEM NOTATION

Notations and problem presentation will be similar as in [13]. Let us assume that the number of memory banks is limited by an electronic device, and that there are \( m \) internal memory banks with limited capacity, denoted by \( c_j \), \( 0 < c_j < \infty \) \((j = 1, \ldots, m)\), and one external memory bank whose size is supposed to be large enough so that it can be considered as unlimited \((c_0 = \infty)\), i.e. the chosen memory architecture is similar to the one of TI C6201 device. Size of a memory bank is given in kilo Bytes (kB), and each memory bank is denoted by \( \beta_j \), \( j = 0, \ldots, m \).

Without loss of generality, let us assume that a particular application is written in C++ code as a set of binary operations between the involved data structures. The term data structure is used for scalars, arrays, or similar structures of the applications. Further, let us assume that a set of all data structures is finite and depends on a program code. Furthermore, let us mark each data structure with \( \alpha_i \), where \( i = 1, \ldots, n \), where \( n \) is a number of data structures. Data structure \( \alpha_i \) size is defined in kB and marked by \( a_i \). Due to the fact that external memory is a huge mass storage, all operations with data structures will be \( p \) times slower if mapped to the external memory bank. Therefore, we say that \( p \) is a penalty cost for loading data structure from the external memory bank. As all operations are not used at the same time, program execution time \( T \) is divided into \( t \) time blocks, \( I_t \); it is not necessary that time blocks have constant size, and it is allowed to reconsider and move data structures before and after each time block. For instance, we say that a time block has size equal to one if it contains only one operation. If all time blocks are of size one, moving data structures is allowed before and after each operation. Since the access to data structures during the particular time block can differentiate, we define its access cost in milliseconds per kilo Bytes (ms/kkB) and mark access cost to data structure \( \alpha_i \) at time block \( I_t \) by \( e_{it} \).

As previously mentioned, given that the parallel data structure usage can speed up program execution, moving data structure will be useful, especially because memory banks are of limited size. We assume that moving data structure is allowed only in between two time blocks and not during, without loss of generality. The problem with only one time block is a static memory allocation problem and more about it can be found in [8] and [12]. Moving data structures and rearranging their allocation is useful because in some cases moving data structure from external to internal memory bank and loading it from the internal memory bank can be less expensive than loading and operating with it while it
is mapped to the external memory bank. Every data structure movement is also calculated.

It is assumed that the data structure movement between two internal memory banks costs $l$ times its size, while the movement between internal and external memory banks is $v$ times its size.

Similarly, when data structures are jointly involved in the same operation, i.e. have to be accessed at the same time, it will be assumed that data structure location affects its loading time. If two data structures, say $a_s$ and $a_d$, are jointly involved in the same operation during the particular time block $I_t$, the given pair is called "conflict pair" with notation $[a_s, a_d]$. The number of conflict pairs at time block $I_t$, $t = 1, \ldots, T$ is denoted by $k_t$, while the set of all conflict pairs accessed at time block $I_t$ is given by $O_t$. Accessing the conflict pair $[a_s, a_d]$, at time block $I_t$ will cost $b_{a_s,a_d,t}$ if both data structures are mapped onto two different internal memory banks, $2b_{a_s,a_d,t}$ if both data structures are mapped onto the same internal memory bank, respectively; $pb_{a_s,a_d,t}$ if one data structure is mapped onto the external memory bank, and $2pb_{a_s,a_d,t}$ if both data structures are mapped onto the external memory bank.

Let us assume that set $O_t$ consists of $k_t$ conflict pairs, marked as $[a_{s_1}, a_{d_1}], [a_{s_2}, a_{d_2}], \ldots, [a_{s_{k_t}}, a_{d_{k_t}}]$. In order to reduce the number of indices, for $r$-th conflict pair $[a_{s_r}, a_{d_r}], r = 1, \ldots, k_t$ instead of using notation $b_{a_{s_r},a_{d_r},t}$ conflict cost will be written as $b_{s_r,t}$. The set of all required data structures at time block $t$ is denoted by $P_t$.

Again, it is assumed that all data structures are mapped to the external memory bank at the beginning of the program execution. Also, at the beginning of the program execution, the number of time intervals $(T)$ is given and for each time interval $I_t$, $t = 1, \ldots, T$, sets $P_t$ and $O_t$ are followed with all necessary data (access cost for each data structure $a_i \in P_t$ and conflict costs for each conflict pair $[a_{s_r}, a_{d_r}], r = 1, \ldots, k_t$ in a particular time interval $I_t$).

The first mathematical formulation for DMAP in embedded systems is presented as it was proposed in [13].

For all $(i, j, k) \in \{1, \ldots, n\} \times \{0, \ldots, m\} \times \{1, \ldots, T\}$ a decision variable $x_{i,j,t}$ is set to one if and only if data structure $a_i$ is allocated to the memory bank $b_j$ during the time block $I_t$, $x_{i,j,t} = 0$ otherwise. Given that, the conflict pair is considered as closed if the involving data structures are mapped onto two different memory banks but open otherwise. For all conflicts $r \in \{1, \ldots, k_t\}$ at time block $I_t \in \{1, \ldots, T\}$, decision variable $y_{r,t}$ will be set to one if and only if during the time interval $I_t$ conflict $r$ is closed, otherwise $y_{r,t} = 0$. Further, data structure moving is represented by the two following sets of variables: for all $i \in \{1, \ldots, n\}$ and $t \in \{1, \ldots, T\}$, $\bar{w}_{i,t}$ is set to one if and only if data structure $a_i$ has been moved from a memory bank $b_j \neq b_0$ to a different memory bank $b_{j'} \neq b_0$ between time blocks $I_{t-1}$ and $I_t$. For all $a_i, i = \{1, \ldots, n\}$, at time block $I_t = \{1, \ldots, T\}$, $\bar{w}_{i,t}$ would be set to one if and only if data structure $a_i$ was moved from internal memory bank to the external memory bank, or if it was moved from the external memory bank to an internal memory bank between time blocks $I_{t-1}$ and $I_t$.

Before presenting ILP formulation for DMAP, let us recall all input and output
data described in [13].

Input:

- $T$ - number of time intervals.
- $n$ - number of data structures.
- $m$ - number of internal memory banks.
- $p$ - penalty cost for operating with data structure when it is mapped to the external memory bank.
- $v$ - penalty cost for data structure movement from external memory bank to the internal and vice versa.
- $l$ - penalty cost for data structure movement between two different internal memory banks.
- $a_i$ - size of data structure $\alpha_i, i = 1, \ldots, n$.
- $c_j$ - size of internal memory bank $\beta_j, j = 1, \ldots, m$.
- $P_t$ - set of data structures which are going to be used at the time block $I_t, t = 1, \ldots, T$.
- $e_{ij,t}$ - number of times that $\alpha_i \in P_t, i = 1, \ldots, |P_t|$ was accessed during the time block $I_t, t = 1, \ldots, T$.
- $O_t$ - set of conflict pairs for time block $I_t, |O_t| = k_t, t = 1, \ldots, T$.
- $b_{ij,t}$ - conflict cost for conflict pair $\{\alpha_{i_r}, \alpha_{i_d}\} \in O_t, r = 1, \ldots, k_t$ for time block $I_t, t = 1, \ldots, T$.

Output:

- $x_{ij,t}$ - the decision if data structure $\alpha_i, (i = 1, \ldots, n)$ is to be mapped to the memory bank $\beta_j, (j = 0, \ldots, m)$ during the time interval $I_t, (t = 1, \ldots, T)$.
- $y_{ir,t}$ - the decision if conflict $r$ is to be closed during time interval $I_t, (t = 1, \ldots, T)$.
- $w_{ij,t}$ - the decision if data structure $\alpha_i, i = 1, \ldots, n$ is to be moved from one internal memory bank to a different internal memory bank between time intervals $I_{t-1}$ and $I_t$.
- $w'_{ij,t}$ - the decision if data structure $\alpha_i, (i = 1, \ldots, n)$ is to be moved from the external to the internal memory bank or vice versa between time intervals $I_{t-1}$ and $I_t$. 
Now, DMAP formulation known from the literature can be described as follows:

$$\text{min } f = \sum_{i=1}^{T} [(p - 1) \sum_{a \in P_i} (e_{i,t} \cdot x_{i,\beta_i}) - \sum_{r=1}^{k_i} (y_{r,t} \cdot b_{r,j}) + \sum_{a \in P_i} a_i(l \cdot w_{i,t} + v \cdot w'_{i,t})] \quad (1)$$

Subject to the constraints

$$\sum_{j=0}^{m} x_{i,j,t} = 1, \quad i \in [1, ..., n], \quad t \in [1, ..., T] \quad (2)$$

$$\sum_{a \in P_i} x_{i,j,t} \cdot a_i \leq c_j, \quad j \in [1, ..., n], \quad t \in [1, ..., T] \quad (3)$$

$$x_{s,j,t} + x_{d,j,t} \leq 2 - y_{r,t} \quad (a_s, a_d) \in P_i, \quad j \in [0, ..., n], \quad r \in [1, ..., k], \quad t \in [1, ..., T] \quad (4)$$

$$x_{i,j,t-1} + x_{i,j',t} \leq 1 + w_{i,t} \quad i \in [1, ..., n], \quad (j, j') \in [1, ..., m]^2, \quad (\beta_j \neq \beta_{j'}), \quad t \in [1, ..., T] \quad (5)$$

$$x_{i,0,t-1} + x_{i,j,t} \leq 1 + w'_{i,t} \quad i \in [1, ..., n], \quad j \in [1, ..., m], \quad t \in [1, ..., T] \quad (6)$$

$$x_{i,j,t-1} + x_{i,0,t} \leq 1 + w'_{i,t} \quad i \in [1, ..., n], \quad j \in [1, ..., m], \quad t \in [1, ..., T] \quad (7)$$

$$x_{i,j,0} = 0 \quad i \in [1, ..., n], \quad j \in [1, ..., m] \quad (8)$$

$$x_{i,j,0} = 1 \quad i \in [1, ..., n] \quad (9)$$

$$x_{i,j,t} \in [0, 1] \quad i \in [1, ..., n], \quad j \in [1, ..., m], \quad t \in [1, ..., T] \quad (10)$$

$$w_{i,t} \in [0, 1] \quad i \in [1, ..., n], \quad t \in [1, ..., T] \quad (11)$$

$$w'_{i,t} \in [0, 1] \quad i \in [1, ..., n], \quad t \in [1, ..., T] \quad (12)$$

$$y_{r,t} \in [0, 1] \quad r \in [1, ..., k], \quad t \in [1, ..., T] \quad (13)$$

Constraints (2) state that every data structure is allocated to only one memory bank at time interval $I_t$. Constraints (3) ensure that the total size of all data structures allocated to any memory bank at time interval $I_t$ does not exceed its size. Constraints (4) - (7) ensure that variables $y_{r,t}, w_{i,t}$ and $w'_{i,t}$ are set appropriately. The initial conditions are given by the constraints (8) and (9) and finally, constraints (10)-(13) enforce binary requirements.

Note that the presented mathematical formulation does not correspond to the DMAP problem. More precisely, calculation of the cost function is based only on the fact that data structures are mapped to the same memory bank, though, both data structures can be mapped to the external memory bank or to the same internal memory bank. Additionally, the cost of accessing data structure $a_i$ during time interval $I_t$ is calculated as $(p - 1)e_{i,j}$ if the data are mapped to the external memory bank and as 0 if the data are mapped to the internal memory bank, which is not correct. We believe that $\sum_{i=1}^{T} \sum_{a \in P_i} e_{i,j}$ is unintentionally left out, given the fact that with this constant, the cost function (1) indeed corresponds to the DMAP formulation.
problem. Still, the way the solution to the DMAP problem was calculated in [11], [13] and [15] included the mentioned oversights.

Considering that it was not easy to notice that the mentioned constant was left out, we have given an improved ILP formulation for the DMAP in embedded system in the following section.

3. IMPROVEMENT TO THE EXISTING ILP FORMULATION FOR DMAP IN EMBEDDED SYSTEMS

Let binary variables $x_{i,j,t}$, $w_{i,j}$ and $w'_{i,j}$ ($i = 1, ..., n$, $j = 0, ..., m$, $t = 1, ..., T$) have the same meaning as before, and let binary variables $y_{r,t}$, defined such that $r$ represent a conflict pair $(α_{s, r}, α_{d, r})$ at time block $I_t$, have slightly different meaning, i.e.

$$y_{r,t} = \begin{cases} 
1, & \text{if data } \alpha_{s, r} \text{ and } \alpha_{d, r} \text{ are both mapped into the same memory bank at time block } I_t \ (α_{s, r}, α_{d, r}) \in I_t, t = 1, ..., T \ (14) \\
0, & \text{otherwise}
\end{cases}$$

Additionally, let $e_{it}$ represent the cost for accessing data $α_i$ during the time block $I_t$ instead of denoting the number of that data structure $α_i$ accessed during the time block $I_t$. Value of the $e_{it}$ will actually differ, concerning the previous definition by the conflict cost where it is involved during time interval $I_t$. We think that with this definition, loading each data structure and accessing the same data structure during the conflict, in which it is involved, are defined more precisely.

Now, the improved formulation for DMAP can be defined as follows:

$$\min f = \sum_{t=1}^{T} \left( \sum_{α_i \in P_t} \left( 1 + (p-1)x_{i,0,t} \right) e_{i,t} + \sum_{r=1}^{k} b_r \left( 1 + y_{r,t} + (p-1)(x_{s,r,t} + x_{d,r,t}) \right) + \sum_{i=1}^{n} a_i \left( lw_{i,t} + vw'_{i,t} \right) \right) \ (15)$$

subject to constraints:

$$x_{s,j,t} + x_{d,j,t} \leq y_{r,t} + 1 \ (α_{s, r}, α_{d, r}) \in P_t, \ t = 1, ..., T \ (16)$$

$$y_{r,t} \in \{0, 1\} \ (17)$$

and constraints (2), (3), (5)-(12) from the existing ILP formulation.

Constraints (16) correspond to the constraints (4) in accordance with the new definition of variables $y_{r,t}$.

Access cost at particular time block $I_t$ is equal to $\sum_{α_i \in P_t} \left( 1 + (p-1)x_{i,0,t} \right) e_{i,t}$, which covers cases when data structure is mapped to the external memory bank. At the same time block, expression $1 + y_{r,t} + (p-1)(x_{s,r,t} + x_{d,r,t})$ is equal to 1 if data structures from the same conflict pair are mapped onto two different internal memory banks; it is equal to 2 if they are mapped onto the same internal memory.
bank, respectively, and it is equal to $p$ if one conflict data is mapped to the external memory bank and is equal to $2p$ if both data structures are mapped to the external memory bank. Therefore,

$$\sum_{r=1}^{k} b_{r,t} \left( 1 + y_{r,t} + (p - 1)(x_{s,r,0,t} + x_{d,r,0,t}) \right)$$

correspond to conflict cost at time interval $I_t$, while movement cost between two time blocks is again calculated by the sum

$$\sum_{t=1}^{T} \sum_{i=1}^{n} a_i \left( lw_{i,t} + vw'_{i,t} \right)$$

Now, given that all oversights are covered by the new proposed cost function, the improved ILP formulation corresponds to the proposed DMAP in embedded systems.

For $T = 1$, DMAP formulation becomes ILP formulation, which was proposed and proven to be feasible for static version of the memory allocation problem in [8]. Let us illustrate DMAP and its ILP formulation on a short example.

An illustrative example is given on a small size instance.

**Example 3.1.** There are 9 data structures ($\alpha_i$, $i = 1, \ldots, 9$), which have to be placed into 2 internal memory banks ($\beta_1$ and $\beta_2$), each size of 1000 (kB) ($c_j = 1000$, $j = 1, 2$), and 1 external memory bank ($c_0 = \infty$). Sizes of data structures are given in Table 1. Loading time is divided into three time blocks of specified sizes. Data structures are used in pairs. Access cost for each data structure at each time block, together with the conflict sets are given in Table 2. For instance, at time block $I_1$, 4 data structures are used, $P_1 = \{\alpha_2, \alpha_5, \alpha_6, \alpha_9\}$, with zero access time each and with two conflict pairs. The first one is between data structures $\alpha_5$ and $\alpha_9$ ($k_{1,1} = (\alpha_5, \alpha_9)$) with conflict cost $b_{1,1} = 592$, and the second one is between data structures $\alpha_2$ and $\alpha_3$ ($k_{2,1} = (\alpha_2, \alpha_3)$) with conflict cost $b_{2,1} = 192$. Loading all operations from the external memory bank costs $p = 16$ times more than loading them from the internal memory bank. Moving data between two internal memory banks costs $l = 1$ (ms/kB), and moving between internal and external memory bank $v = 4$ (ms/kB).

Let us assume that at the beginning of the program ($t = 0$), all data structures are mapped to the external memory bank. Knowing the order of the data structure usage, we can move them before each time block in order to reach better results. Table 3 represents the optimal solution obtained by using IBM ILOG CPLEX optimization solver for a mathematical model presented above. Solution can be interpreted as follows: at time block $I_1$, data structures $\alpha_1, \alpha_3, \alpha_4, \alpha_6$, and $\alpha_7$ should be mapped to the external memory bank, marked as $\beta_0$, while data structures $\alpha_5$ and $\alpha_9$ should be mapped to the memory bank marked as $\beta_1$, and finally, data structures $\alpha_2$ and $\alpha_8$ should be mapped to the memory bank marked as $\beta_2$. Then, total execution time, which consists of moving time and conflict time for each
as follows: min

\[ f(x) \] is replaced with data structure \( \alpha \) and moved from memory allocation \( \beta_2 \) to the memory allocation \( \beta_0 \) and the other way around, moving cost is equal to \((256 + 256) \cdot 4 = 2048\) (ms). Further, conflict costs for the same block is equal to \(1,024 + 1,024 + 64 \cdot 2 = 2,176\) (ms), which brings total block cost to 4,224 (ms). Combining the results for all three time blocks gives the problem execution cost of 17,772 (ms).

| Table 1: Data structure size |
|-------------------------------|
| data structures | \( a_1 \) | \( a_2 \) | \( a_3 \) | \( a_4 \) | \( a_5 \) | \( a_6 \) | \( a_7 \) | \( a_8 \) | \( a_9 \) |
| size of data struct. | 256 | 4 | 496 | 256 | 4 | 256 | 256 | 256 | 256 |

| Table 2: Data access and conflict costs for \( t = 1 \) (left), \( t = 2 \) (middle) and \( t = 3 \) (right) |
|--------------------------------------------------|
| **Conflicts** | **Access c.** | **Cost** |
| \( k_{1f} \) | \( a_2 \) | \( a_5 \) | 592 |
| \( k_{2f} \) | \( a_2 \) | \( a_5 \) | 192 |
| \( k_{3f} \) | \( a_2 \) | \( a_5 \) | 192 |
| **Time block:** 1 | **Time block:** 2 | **Time block:** 3 |
| Data set | \( a_2 \) | \( a_5 \) | \( a_9 \) | \( a_9 \) | \( a_9 \) | 592 |
| Access c. | 4 | 256 | 256 | 4 | 496 | 256 | 256 | 256 | 256 |
| Conflicts | \( a_2 \) | \( a_5 \) | \( a_2 \) | \( a_5 \) | \( a_2 \) | \( a_5 \) | 576 | \( a_4 \) | \( a_7 \) | 1,024 |
| Conflicts | \( a_1 \) | \( a_4 \) | \( a_1 \) | \( a_4 \) | \( a_4 \) | 64 | \( a_4 \) | \( a_7 \) | 1,024 |
| Conflicts | \( a_1 \) | \( a_3 \) | 1,024 | \( a_4 \) | \( a_4 \) | 64 |

Table 3: Solution to the example for DMAP formulation

| Mem. allocation | \( t = 0 \) | \( t = 1 \) | \( t = 2 \) | \( t = 3 \) |
|-----------------|-------------|-------------|-------------|-------------|
| \( \beta_0 \) | All data | \( a_1, a_3, a_4, a_6, a_7 \) | \( a_6, a_8 \) | \( a_1, a_8 \) |
| \( \beta_1 \) | \( a_2 \) | \( a_5, a_8 \) | \( a_3, a_5, a_7 \) | \( a_3, a_5, a_7 \) |
| \( \beta_2 \) | \( a_2 \) | \( a_9 \) | \( a_1, a_2, a_4, a_9 \) | \( a_2, a_4, a_6, a_9 \) |

Table 4: Costs

| | \( t = 1 \) | \( t = 2 \) | \( t = 3 \) |
|-----------------|-------------|-------------|-------------|
| Access cost | 520 | 1,012 | 768 |
| Moving cost | 2,080 | 6,080 | 2,048 |
| Conflict cost | 784 | 2,304 | 2,176 |
| Total cost per block | 3,384 | 9,396 | 4,992 |

Now, the solution in terms of the proposed ILP formulation can be interpreted as follows: \( \min f = 17,772 \) with nonzero decision variables \( x_{1,0,1}, x_{3,0,1}, x_{4,0,1}, x_{6,0,1}, x_{7,0,1}, x_{8,1,1}, x_{8,5,1}, x_{8,1,2}, x_{8,2,1}, x_{9,0,2}, x_{9,2,1}, x_{3,1,2}, x_{3,5,2}, x_{7,1,2}, x_{1,2,2}, x_{2,2,2}, x_{4,2,2}, x_{9,2,2}, x_{1,0,3}, x_{8,0,3}, x_{3,1,3}, x_{3,2,3}, x_{7,2,3}, x_{2,3,3}, x_{4,2,3}, x_{6,2,3}, x_{9,2,3}, y_{1,2}, y_{2,2}, y_{3,2}, w_{2,1}, w_{5,1}, w_{8,1}, w_{9,1}, w_{1,2}, w_{3,2}, w_{4,2}, w_{7,2}, w_{8,2}, w_{1,3}, \) and \( w_{6,3} \).

In order to compare the presented ILP formulation for the DMAP with the one known from the literature, both formulations were coded in C++ and tested on Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz, 2394MHz, 4 Core(s) with 8GB
RAM with CPLEX 12.6. For experimental testings of the proposed ILP formulations we used the same set of instances as the one reported by Soto et. al in [13]. The set of instances can be downloaded under the name dmap.zip from http://www.optsicom.es/dmap/dmap.zip. More details about these instances are given in Section 5.

In these testings, penalty cost \( p \) is set to be equal to 16ms/kB, movement data structure cost from internal to external memory bank \( (v) \) and vice versa is set to be equal to 4ms/kB, while the movement between two internal memory banks \( (l) \) is set to be equal to 1ms/kB. Comparison of the presented Soto et al. [13] formulation and our formulation is presented in Table 5. The name of an instance is given in the first and the fifth column. Solution value obtained for the particular instance by using ILP formulation presented in [13] is given in the second and the sixth column. Solution values for the corrected cost function of the same ILP formulation, as described at the end of the Section 2, are given in the third and the seventh column. Finally, solution values of the tested instances by using our formulation are given in the fourth and the eight column. Time limit for all testings was set to 7200 sec. Sign "*" was used in case that testings were stopped earlier because of status "out of memory". On majority of instances, the optimal solution value was found using both formulations. On instances such as "myciel4dy.col"-"r125.1dy.col", the found solutions were not proved to be the optimal. As it can be seen from Table 5, on instances where optimal solution was not reached, CPLEX was able to find better results by using our ILP formulation on 4 instances and on 1 by using Soto et al. [13] corrected formulation.

4. VNS FOR THE PROPOSED DMAP PROBLEM

The problem of combinatorial optimization, presented in this paper, can be solved by using exact methods, but because of its number of variable limitations for large scale variables, metaheuristics are more useful. Therefore, DMAP was solved by using VNS hybridized with Tabu Search (TS) in [13] and GRASP with ejection chains in [11]. In this paper, a VNS metaheuristic method hybridized with Variable Neighborhood Descent (VND) search, which corresponds to the proposed DMAP model, is presented. VNS metaheuristic was first presented by Mladenović and Hansen in [9], and the main idea was proposed in 1995, [10]. Later, it was followed with several papers aimed at improving the method, [2, 3, 4, 5, 6], and [7]. The main idea of this heuristic is that starting from one initial solution by using systematic search, we move to the solution that is locally the best.

**Solution space.** Each solution can be represented by a matrix (two-dimensional array) \( Z \) with \( T \) rows and \( n \) columns. Value of element \( z_{ti} \) represents a label of a memory bank on which the data structure \( \alpha_i \) is allocated at the time interval \( I_t \) (\( t = 1, 2, ..., T \), \( i = 1, 2, ..., n \)). Only feasible solutions are considered, i.e. data structures are allocated in such a way that there are no overloaded banks. In other words, for each time interval \( I_t \) and each bank \( \beta_j \), the following condition is
Table 5: Comparison of the ILP formulation for the DMAP presented in Soto et al [13], their formulation with correction and our ILP formulation

| Instance       | from Soto et al [13] without correct val. | correct val. formulat. | from Soto et al [13] without correct val. | correct val. formulat. |
|----------------|-------------------------------------------|------------------------|-------------------------------------------|------------------------|
|                | Instance | val | val | val | Instance | val | val | val | Instance | val | val | val |
| adpcmdy        | 37366    | 44192 | 44192 | mulsol2dy        | * | * | * | mulsol2dy        | * | * | * |
| alidydy        | * | 107518 | 107518 | mulsol4dy        | * | * | * | mulsol4dy        | * | * | * |
| cjpegdy        | -423727 | 4466800 | 4466800 | mulsol5dy        | * | * | * | mulsol5dy        | * | * | * |
| compressdy     | -222272 | 342592 | 342592 | myciel3dy        | * | 6379 | 6379 | myciel3dy        | * | 6379 | 6379 |
| fpsol2i2dy     | * | * | * | myciel4dy        | * | 18225 | 18295 | myciel4dy        | * | 18225 | 18295 |
| fpsol2i3dy     | * | * | * | myciel5dy        | * | 40380 | 40380 | myciel5dy        | * | 40380 | 40380 |
| gsmnewdy       | 7320    | 7808 | 7808 | myciel6dy        | * | 108726 | 108726 | myciel6dy        | * | 108726 | 108726 |
| gsmdy          | 548645.75 | 1353390 | 1353390 | myciel7dy        | * | * | * | myciel7dy        | * | * | * |
| gsmdycorrdy    | -312628 | 494119 | 494119 | queen5dy         | * | 21859 | 21859 | queen5dy         | * | 21859 | 21859 |
| inithx1i1dy    | 0 | * | * | queen6dy         | * | 43306 | 39144 | queen6dy         | * | 43306 | 39144 |
| lmsbdy         | -7245892 | 7409669 | 7409669 | queen7dy         | * | 88652 | 85606 | queen7dy         | * | 88652 | 85606 |
| lmsbdy01dy     | -4051636 | 4350640 | 4350640 | queen8dy         | * | * | 138399 | queen8dy         | * | * | 138399 |
| lmsbdvdy       | -4061214 | 4332294 | 4332294 | r125.1cdy        | * | * | * | r125.1cdy        | * | * | * |
| lmsbdvdyexpdy  | -4035252 | 4367024 | 4367024 | r125.1dy         | * | 62354 | 60931 | r125.1dy         | * | 62354 | 60931 |
| lpcdy          | 23802    | 26888 | 26888 | r125.5dy         | * | * | * | r125.5dy         | * | * | * |
| mpeg2enc2dy    | 7788.8586 | 9812.312 | 9812.312 | spectraldy       | 6352 | 15472 | 15472 | spectraldy       | 6352 | 15472 | 15472 |
| mpegdy         | 5805.625 | 10613.63 | 10613.63 | trellisdy        | 1331.563 | 1805.563 | 1805.563 | trellisdy        | 1331.563 | 1805.563 | 1805.563 |
| mug100_1dy     | 12797    | 29847 | 29847 | turbocodedy      | 847 | 3195 | 3195 | turbocodedy      | 847 | 3195 | 3195 |
| mug100_25dy    | 11621    | 28429 | 28429 | volterra3dy      | 166 | 178 | 178 | volterra3dy      | 166 | 178 | 178 |
| mug88_1dy      | 10227    | 25305 | 25305 | zeroin1dy        | * | * | * | zeroin1dy        | * | * | * |
| mug88_25dy     | 9157     | 24181 | 24181 | zeroin2dy        | * | * | * | zeroin2dy        | * | * | * |
| mulsol1i1dy    | * | * | * | zeroin3dy        | * | * | * | zeroin3dy        | * | * | * |
satisfied: 
\[ \sum_{i \in \mathcal{F}} a_i \leq c_j. \]

Note that the above expression represents the total size of memory bank \( \beta_j \) occupied by data structures allocated to that bank during the time interval \( I_t \).

Similarly, a solution can be represented by a matrix \( X \), where \( x_{i,j,t} \in X \) is equal to one if data \( \alpha_i \) is allocated to the memory bank \( \beta_j \) at the time block \( I_t \) and zero otherwise. For a matrix \( Z \), which represents the solution, it is possible to evaluate the solution \( X \) and its respective objective function value \( f(X) \). The objective function value consists of two parts:

- Costs of moving data structures from one memory bank to another,
- Costs of accessing conflict pairs.

Complexity of computing the first part of the objective function is \( O(nT) \), while the cost of computing the second part is \( O(K) \), where \( K \) represents a total number of conflict pairs (\( K = \sum_{t=1}^{T} k_t \)).

In the solution space, we introduce two types of moves:

- Insertion move,
- Swap move.

**Insertion move** consists of picking one time interval \( I_t \) and one data structure \( \alpha_i \), and moving that data structure from the bank \( \beta_{i_1} \) on which it is allocated in the current solution to the bank \( \beta_{j_2} \) \( (j_2 \neq j_1) \). Understandably, it is necessary to choose \( \beta_{j_2} \) in such a way that the new solution obtained by this move is feasible. If a solution is represented by a matrix \( Z \), this move consists of setting value of element \( z_{i,j,t} \) to \( j_2 \).

**Swap move** consists of picking time interval \( I_t \), two data structures \( \alpha_{i_1} \) and \( \alpha_{i_2} \) currently allocated to different banks and exchanging their allocations. If a solution is represented by a matrix \( Z \), this move consists of exchanging values of elements \( z_{i_1,j,t} \) and \( z_{i_2,j,t} \).

Hence, based on the introduced moves, we can define neighborhoods of the solution space. So, neighborhood \( \mathcal{N}_k(X) \) consists of all solutions \( X' \) obtained by applying \( k \) successive moves (Insertion or Swap move) starting from the solution \( X \). Also, we introduce two neighborhoods, \( \mathcal{N}_{ins}(X) \) and \( \mathcal{N}_{swap}(X) \), as the sets of all solutions obtained by applying Insertion move or Swap move (respectively) on solution \( X \). Note that the union of neighborhoods \( \mathcal{N}_{ins} \) and \( \mathcal{N}_{swap} \) represents the neighborhood \( \mathcal{N}_1 \).

Now, VNS heuristic can be defined in such a way that starting from initial feasible solution \( X \) it “shakes” that solution by creating another feasible solution \( X' \in \mathcal{N}_k(X) \) and applies local search method in order to move to a better solution \( X'' \). If such a solution is not better than the current incumbent \( X \), we create another
neighborhood set $N_{k+1}(X)$ and seek for a better solution until $k$ reaches its maximum $k_{\text{max}}$. If $X''$ is better than a current incumbent solution, it becomes new incumbent and $k$ becomes $k_{\text{min}}$. This systematic local environment change is needed because of the fact that local minimum within one environment does not have to be local minimum of the other environment, but a global minimum is a local minimum relative to all environments. Changing environments also enables to get out from the local minimum. VNS metaheuristic is illustrated with Algorithm 1.

**Algorithm 1: Variable Neighborhood Search metaheuristics**

```plaintext
1 $X \leftarrow \text{InitialSolution}();
2 X_{\text{opt}} \leftarrow X;
3 f_{\text{opt}} \leftarrow f(X);
4 \text{repeat}
5 \quad k \leftarrow k_{\text{min}};
6 \quad \text{repeat}
7 \quad \quad X' \leftarrow \text{Shake}(X,k);
8 \quad \quad X'' \leftarrow \text{LocalSearch}(X');
9 \quad \quad \text{if } f(X'') < f_{\text{opt}} \text{ then}
10 \quad \quad \quad X \leftarrow X'';
11 \quad \quad \quad f_{\text{opt}} \leftarrow f(X);
12 \quad \quad \quad k \leftarrow k_{\text{min}};
13 \quad \quad \text{else}
14 \quad \quad \quad k \leftarrow k + k_{\text{step}};
15 \quad \quad \end{\text{if}}
16 \quad \text{until } k > k_{\text{max}};
17 \text{until StoppingCondition();}
```

*Initial solution.* At the beginning, all data structures are mapped to the external memory bank. After that, for each data structure $\alpha_i$ and each time interval $I_t$, we try to “reallocate” $\alpha_i$ to one of the banks such that the obtained solution is feasible and better than the incumbent. The pseudocode for calculation of an initial solution is given in Algorithm 2.

*Shaking.* Shaking in Neighborhood $N_k (\text{Shake}(X,k))$ consists of performing $k$ randomly selected moves. For each of these $k$ moves, Insertion move or Swap move were chosen equally (with probability equal to $p = 0.5$). The pseudo code for Shaking procedure is given in Algorithm 3.
Algorithm 2:

1 Function InitialSolution();
2 for ← 1 to n do
3     for t ← 1 to T do
4         X_{t,i} ← 0;
5     end
6 end
7 f_c ← f(X);
8 for i ← 1 to n do
9     for t ← 1 to T do
10        for j ← 1 to m do
11           x_0 ← X_{t,i};
12           X_{t,j} ← j;
13           if Feasible(X) and f(X) < f_c then
14               f_c ← f(X);
15           else
16               X_{t,i} ← x_0;
17           end
18        end
19 end
20 return X';

Algorithm 3:

1 Function Shake(X, k);
2 X' ← X;
3 for i ← 1 to k do
4     p ← RandomNumber(0, 1);
5     if p ≤ 0.5 then
6         InsertMethod:
7             repeat
8                 t_r ← RandomNumber(1, T);
9                 i_r ← RandomNumber(1, n);
10                repeat
11                    j_r ← RandomNumber(0, m);
12                until j_r ≠ CurrentPosition(i_r, t_r);
13             until NotFeasibleSolution;
14             Move data structure α_i to the memory location β_j at time block I_{t_r}.
7 else
8     SwapMethod:
9         repeat
10            t_r ← RandomNumber(1, T);
11            i_{r_1} ← RandomNumber(1, n);
12         repeat
13             i_{r_2} ≠ i_{r_1},
14             until i_{r_2} ≠ i_{r_1};
15         until NotFeasibleSolution;
16         Swap memory locations for data structures α_{i_{r_1}} and α_{i_{r_2}} at time block I_{t_r}.
21 end
26 end
Local Search is implemented as the Variable Neighborhood Descent (VND) method based on two previously defined moves (and corresponding neighborhoods $N_{\text{ins}}$ and $N_{\text{swap}}$). This means that the neighborhood $N_{\text{ins}}(X')$ of a current solution $X'$ is examined in order to find a solution $X'_1$, which is better than the solution $X'$ ($f(X'_1) < f(X')$). If such a solution $X'_1$ exists, the corresponding move is performed and examining the neighborhood $N_{\text{ins}}$ of the new solution continues. If there is no better solution in the neighborhood $N_{\text{ins}}$ of the incumbent, examining the neighborhood $N_{\text{swap}}$ of the incumbent continues. If there is a solution $X'_1 \in N_{\text{swap}}(X')$, the corresponding move is performed and examining the neighborhood $N_{\text{swap}}$ of the new solution continues. If during examination of neighborhoods $N_{\text{swap}}$ at least one improvement is made, we return to neighborhood $N_{\text{ins}}$, otherwise local search (VND) is finished. The pseudo-code of VND and included functions are given in Algorithms 4, 5, and 6.

**Algorithm 4: Function for Variable Neighborhood Descent**

```latex
1 Function VND(X);
2 \hspace{1em} k \leftarrow 1;
3 \hspace{1em} X' \leftarrow X;
4 while k \leq 2 do
5 \hspace{2em} if k = 1 then
6 \hspace{3em} X'' \leftarrow \text{LSIns}(X);
7 \hspace{4em} if f(X'') < f(X') then
8 \hspace{5em} X' \leftarrow X'';
9 \hspace{4em} k \leftarrow 1;
10 \hspace{3em} else
11 \hspace{4em} k \leftarrow 2;
12 \hspace{2em} end
13 \hspace{2em} else
14 \hspace{3em} X'' \leftarrow \text{LSSwap}(X);
15 \hspace{4em} if f(X'') < f(X') then
16 \hspace{5em} X' \leftarrow X'';
17 \hspace{4em} k \leftarrow 1;
18 \hspace{4em} else
19 \hspace{5em} k \leftarrow 3;
20 \hspace{4em} end
21 \hspace{2em} end
22 end
23 return X';
```
Algorithm 5: Function for local search in neighborhood $N_{\text{ins}}$

1. **Function** $LSIns(X)$;
2. $X' \leftarrow X$;
3. **for** $i \leftarrow 1$ **to** $n$ **do**
4.    **for** $t \leftarrow 1$ **to** $T$ **do**
5.       $X'' \leftarrow \text{Move}(X', i, t)$;
6.       **if** $\text{Feasible}(X'')$ **and** $f(X'') < f(X')$ **then**
7.          $X' \leftarrow X''$;
8.       **end**
9.    **end**
10.   **end**
11.   **return** $X'$;

Algorithm 6: Function for local search in neighborhood $N_{\text{swap}}$

1. **Function** $LSSwap(X)$;
2. $X' \leftarrow X$;
3. **for** $i_1 \leftarrow 1$ **to** $n$ **do**
4.    **for** $i_2 \leftarrow i_1 + 1$ **to** $n$ **do**
5.       **for** $t \leftarrow 1$ **to** $T$ **do**
6.          $X'' \leftarrow \text{Swap}(X', i_1, i_2, t)$;
7.          **if** $\text{Feasible}(X'')$ **and** $f(X'') < f(X')$ **then**
8.            $X' \leftarrow X''$;
9.       **end**
10.    **end**
11.   **end**
12.   **return** $X'$;

Let us discuss briefly the complexity of the proposed local search. Cardinality of the neighborhood $N_{\text{ins}}(X')$ of the solution $X'$ is $nmT$ (each of $n$ data structures can be reallocated in each of $T$ time intervals to one of the remaining $m$ memory banks). On the other hand, examination of all neighboring solutions $X'_1$ (suppose that $X'_1$ is obtained by moving data structures $\alpha_i$ from bank $\beta_{i_1}$ to bank $\beta_{i_2}$ during the time interval $I_t$) consists of calculating changes in objective function value.

The change of objective function value consists of

- change in the part of the objective function containing cost of transferring data structure from one bank to another (can be calculated in $O(1)$).

- change in the part of the objective function containing costs of accessing conflicts (can be calculated in $O(k_{i,j})$, where $k_{i,j}$ is number of conflicts involving data structure $\alpha_i$ in time interval $I_t$)

Obviously, the second part depends on a number of conflicts involving specified data structure and specified time interval. However, it is possible to make
aggregate complexity analysis if we note that each conflict participates in change
of the objective function for at most $2m$ neighboring solutions (from neighborhood
$N_{ins}$). Finally, the total complexity for calculating the change of the second part
of the objective function is $O(Km)$, while the total complexity for both parts is
$O(nmT + Km)$. 

Regarding neighborhood $N_{swap}$, cardinality of neighborhood is $\binom{n}{2}T$ (each of $\binom{n}{2}$
pairs of data structures can exchange allocation on each of the $T$ time intervals).
Similarly, calculating change in objective function contains two parts: change in
cost of moving data structure (can be calculated in $O(1)$ time), and change of cost
for conflict. Each conflict participates in change of the objective function for at
most $2(n - 2) + 1$ moves (each of data structures from conflict pair can exchange
allocation with any of the remaining data structures). Now, it can be concluded
that the total complexity is $O\left(\binom{n}{2}T + Kn\right)$.

5. COMPUTATIONAL RESULTS

Experimental results obtained by the proposed VNS algorithm for solving
DMAP are given in this section. VNS heuristic was coded in C++. All computa-
tional experiments were performed on Pentium Core Duo CPU @ 2.66GHz with
6GB RAM. For the experimental testings of the proposed implementation, the set
of instances used is the same as the one reported by Soto in [13].
The set of instances can be downloaded under the name
dmap.zip from http://www.optsicom.es/dmap/dmap.zip.

Instances are classified by their name and three relevant characteristics, notably
time interval, number of data structures, and number of internal memory
banks. In the following table, the main features of the instances are shown: In-
stance name, number of time intervals ($T$), number of data structures ($n$), and
number of available internal memory banks ($m$).
Further, in order to compare with the known results, penalty cost $p$ is set to be equal to 16ms/kB, movement data structure cost from internal to external memory bank ($v$), and vice versa, is set to be equal to 4ms/kB, while movement between two internal memory banks ($l$) is set to be equal to 1ms/kB.

Given that all metaheuritics, including VNS, have a stochastic nature, VNS algorithm was run 20 times for each problem instance. Finishing criteria of the proposed VNS were either time limit or event when $k_{\text{max}}$ was reached. In this implementation, $k_{\text{max}}$ was set to 10, $k_{\text{step}}$ to 1, and $k_{\text{min}}$ to 1. Execution time limit for VNS was set to 7, 200 seconds per instance.

For easier comparison, the results of the presented VNS metaheuristic on tested instances are summarized in Tables 6 - 9, where presented VNS algorithm is compared with IM (from [13]), CPA, GRASP, and GRASP + EC (from [11]) and with BVNS, and RVNS methods (from [16]).

Table 6 is organized as follows. Names of instances, which are sorted alphabetically, are given in the first column. Determined by the solution values presented in columns “from Soto et. al [13] with correct. val.” and “our ILP formulat.” of Table 5 and by the solution values obtained by all considered methods (IM, CPA, GRASP, GRASP + EC, BVNS and RSVNS, and VNS), the best known solution value is presented in the second column. The next six columns are obtained using results from Sevaux et. al [11], where all known methods for solving DMAP (IM, CPA, GRASP and GRASP+EC) till that time were compared. Results in corresponding columns “%dev” are presented as a deviation value from the best known value from the second column (in percentage). Respective running times, which are given only for GRASP and GRASP+EC methods, are copied from [11]. Solution values, respective running times (in seconds), and deviations from the best knowns (in percentage), obtained by VNS metaheuristic proposed in this
Table 6: Comparison between new and existing methods from the literature for solving DMAP

| Instance | Best known | CPA % dev. | GRASP % dev. | GRASP+EC % dev. | VNS % dev. |
|----------|------------|------------|--------------|----------------|------------|
| adlydy   | 107398     | 939.7749   | 149.3545     | 59.5870        | 160.48     |
| cppegdy  | 4466792    | 0.0002     | 1.9302       | 0.0002         | 0.0002     |
| compesly | 34292      | 2.77       | 2.67         | 2.67           | 2.67       |
| fpsel22idy| 258787     | 64.4663    | 33.5800      | 7.9954         | 1015.13    |
| fpsel23idy| 258288     | 60.1959    | 33.7942      | 6.9327         | 1062.37    |
| gsn_newdy| 7984       | 0          | 17295        | 0.01           | 0.01       |
| gsm        | 1355389.87 | <0.0001    | <0.0001      | 0.1300         | 0.22       |
| gsmcyordy | 494118     | 0          | 0.35         | 0.36           | 0.36       |
| inthu1idy | 5652213    | 81.8834    | 24.2327      | 11.1145        | 7.3049     |
| lmsbdy   | 7499660    | 0.0001     | 0.3601       | 0.3301         | 0.3901     |
| lmsbv01idy| 4350588    | 0.0012     | 3.7712       | 1.1312         | 0.8812     |
| lmsbv3idy| 4323116    | 0.0041     | 2.2742       | 0.0041         | 1.1442     |
| lmsbdvyexpdy| 4366972   | 0.0012     | 3.3612       | 2.6312         | 1.8812     |
| lpsedy   | 26888      | 0          | 43.67        | 22.02          | 0.02       |
| mpeg2enc2dy| 9812       | 0          | 45.99        | 9.46           | 0.75       |
| mpegdy   | 10613.625  | 0.15       | 41.75        | 4.62           | 0.13       |
| mug100_1idy| 28890     | 0          | 109.98       | 25.49          | 14.71      |
| mug102_25dy| 28426     | 7.2813     | 13.1467      | 11.89          | 9.4791     |
| mug98_1idy| 25385     | 0.877297   | 94.82432     | 10.18827       | 11.43      |
| mug98_25dy| 24183     | 1.448331   | 74.51606     | 1.448331       | 7.79       |
| mulso1jdy| 459958    | 177.7919   | 223.0343     | 73.8314        | 1096.13    |
| mulso2jdy| 552215    | 130.3367   | 214.9909     | 48.0897        | 106.89     |
| mulso4jdy| 50927     | 132.3606   | 222.5553     | 37.8263        | 1057.35    |
| mulso5jdy| 518854    | 144.7968   | 208.6101     | 40.8412        | 1080.78    |
| mycel3dy | 6379       | 89.14      | 6.88         | 8.42           | 1.24       |
| mycel4dy | 18225      | 46.7084    | 20.2284      | 19.6066        | 6.07       |
| mycel5dy | 39144      | 40.9647    | 29.1557      | 42.4718        | 28.86      |
| mycelidy | 108726    | 65.3172    | 59.49225     | 39.17426       | 94.96      |
| mycel7dy | 411022    | 94.7800    | 106.9965     | 30.7758        | 377.08     |
| mycel9dy | 2151      | 73.7655    | 29.5211      | 29.5211        | 4.76       |
| queen56dy| 39144     | 98.3058    | 50.1604      | 26.9977        | 284        |
| queen66dy| 73264     | 154.6836   | 80.1393      | 45.1697        | 42.82      |
| queen77dy| 133318    | 190.3342   | 74.3438      | 33.4029        | 82.56      |
| r125dy   | 770623    | 203.3532   | 343.4828     | 58.9772        | 700        |
| r125dy   | 60931     | 85.06245   | 19.0191      | 16.75982       | 33.38      |
| r125dy   | 501177    | 203.6588   | 214.9133     | 172.4574       | 1028.86    |
| r125dy   | 501177    | 203.6588   | 214.9133     | 172.4574       | 1028.86    |
| r125dy   | 551100    | 81.5096    | 222.9487     | 47.8225        | 1063.72    |

Table 7 is organized similarly. The first two columns are copied from Table 6. In the next six columns of Table 7, results of testing instances by using BNVS and RVNSs are given. These results appear in groups of three (value, running time (in seconds) and deviation value from the best known value (in percentage)). The individual results (solution value and running time) for each instance, for both algorithms proposed in Sánchez-Oro et. al [16], are provided by Jesús Sánchez-Oro, to whom we are very grateful. Deviation value is calculated as deviation from the Best known solution from the second column, in percentage. The last three columns are copied from Table 6, where the results of the presented VNS are given in the same way as the results for BNVS and RVNSs.
First, we want to point to the fact that with the introduced improved ILP formulation of Soto et. al from [11] and our ILP formulation for DMAP problem in embedded systems, CPLEX obtained better results than all considered methods for 10 instances. Therefore, solution presented in column “Best known” of Tables 6 and 7, is the best solution obtained in comparison with the results presented in Table 5 and solutions obtained by all considered methods (IM, CPA, GRASP, GRASP+EC, BVNS, RSVNS, and VNS).

Table 7: Comparison between new and the most recent methods from the literature for solving DMAP

| Instance | Best known | BVNS | RSNS | VNS |
|----------|------------|------|------|-----|
| value    | time %dev  | value | time %dev | Obj.val. | time %dev |
| adpcm     | 44192      | 44192 | 0.146 | 44192 | 0.0 |
| alidfxy   | 107398     | 107766 | 0.023 | 107658 | 0.471 |
| cipgxy    | 4466792    | 4466792 | 0.105 | 4466799 | 0.973 |
| compressxy| 342952     | 342952 | 0.156 | 342952 | 0.0 |
| fsps0120xy| 2587625    | 261011 | 0.258 | 2625765 | 0.239 |
| gsm_newxy | 2582988    | 2581944 | 0.073 | 2607598 | 0.110 |
| gmdnnewxy | 78080      | 78080 | 0.108 | 78080 | 0.0 |
| lmsbxy    | 1353898.88 | 1353898 | 0.074 | 1353898 | 0.188 |
| lmsbnewxy | 494118     | 494120 | 0.100 | 494118 | 0.920 |
| mnsbxy    | 5652213    | 5652213 | 0.006 | 5654904 | 0.047 |
| mlmsbxy   | 7496600    | 7496600 | 0.030 | 7496600 | 0.032 |
| mlmsbxy   | 435068     | 435068 | 0.040 | 435068 | 0.000 |
| nmmlmsbxy | 4323116    | 4323764 | 0.100 | 4323116 | 0.040 |
| lmsbxy    | 4366972    | 4367004 | 0.009 | 4367004 | 0.052 |
| lpmxy     | 26888      | 26888 | 0.000 | 26888 | 0.040 |
| mpeg2002xy| 9812       | 124632793 | 27.020 | 1091528 | 0.22 |
| mpegxy    | 10613.625  | 106746875 | 1.670 | 1064888 | 0.440 |
| mugl01xy  | 28990      | 30781 | 0.258 | 30515 | 3.628 |
| mugl0025xy| 28429      | 29409 | 0.104 | 28997 | 1.998 |
| mugl78xy  | 25305      | 26048 | 0.153 | 25823 | 0.04 |
| mugl7825xy| 24181      | 24886 | 0.870 | 24569 | 1.666 |
| musol02xy | 459998     | 465557 | 0.000 | 459598 | 0.064 |
| musol04xy | 522115     | 561626 | 0.000 | 557532 | 0.899 |
| musol08xy | 505927     | 511055 | 0.000 | 506162 | 0.000 |
| musol12xy | 518854     | 522975 | 0.000 | 518854 | 0.000 |
| myciel02xy| 6379       | 6382 | 0.000 | 6379 | 0.2 |
| myciel04xy| 18225      | 19041 | 0.122 | 18678 | 0.040 |
| myciel06xy| 39144      | 42224 | 0.000 | 42105 | 3.700 |
| myciel08xy| 108726     | 11446 | 0.000 | 112502 | 0.000 |
| myciel07xy| 411022     | 417987 | 0.000 | 412999 | 0.479 |
| queen05xy | 21151      | 21903 | 0.122 | 21763 | 0.404 |
| queen06xy | 39144      | 41018 | 0.000 | 40722 | 0.000 |
| queen77xy | 73264      | 75493 | 0.000 | 75190 | 0.000 |
| queen85xy | 133130     | 140476 | 0.000 | 137027 | 0.000 |
| r125.05xy | 70625     | 854266 | 0.000 | 836061 | 0.000 |
| r125.15xy | 60951     | 62798 | 0.000 | 62033 | 0.000 |
| r125.50xy | 501175     | 534619 | 0.000 | 528754 | 0.000 |
| spectrally| 15472      | 15472 | 0.000 | 15472 | 0.000 |
| trellisly | 1805.625   | 1896.375 | 0.000 | 1829.812 | 0.000 |
| turbocodedxy| 3195      | 3233 | 0.000 | 3195 | 0.000 |
| volterraxy | 178       | 178 | 0.000 | 178 | 0.000 |
| zeron01xy | 495981     | 494030 | 0.000 | 486921 | 0.000 |
| zeron02xy | 501998     | 515845 | 0.000 | 507314 | 0.000 |
| zeron03xy | 551001     | 572150 | 0.000 | 564324 | 0.000 |

Now, considering the results from Tables 6 and 7, none of the presented methods found all best known solutions. For instance, CPLEX reached 18 of the best known solutions, of 31 solved. Further, IM, CPA, GRASP, and GRASP+EC together were successful in finding 8 of the best known, BVNS 10, RSVNS 20, and
the presented VNS 19. Furthermore, the best known solution obtained only by IM was in two cases (the same number as for the BVNS), the best known solution obtained only by RSVNS was in 9 cases, while the best known solution obtained only by VNS was in 10 cases (same as with CPLEX). CPA, GRASP, and GRASP+EC together have no solution better than the solutions obtained by other considered methods for any of the tested instance. More details about this comparison are given in Table 8.

If we compare presented VNS with IM, CPA, GRASP, and GRASP+EC only (Table 6), we can see that: for 26 instances VNS finds better solutions than the solutions obtained with other four methods; for 12 instances VNS finds solutions equal to the best solutions obtained with other four methods; for 6 instances VNS finds worse solutions than the best solutions obtained with other four methods. Similarly, if we compare solutions of presented VNS with solutions obtained only with BVNS and RSVNS methods (Table 7), we can see that for 21 instances VNS finds better solutions than the solutions obtained with these two methods; for 9 instances solutions equal to the best solutions obtained with these two methods; for 14 instances VNS finds worse solutions than the best solutions obtained with these two methods.

Considering only results of the presented VNS, we can notice that running values are less than 1,800s, which can be considered as very fast execution time. Comparing results of the presented VNS with results from the second column of Tables 6 and 7, we can notice that deviation values varies from < 0.0001% up to 11.2442% for instances where best known solutions are not reached. More precisely, with average deviation from the best solution value of 0.8513%, solution obtained by the proposed VNS differ from the best known solution in less than 1% for 14 instances, between 1% and 2% for 7 instances, and differ more than 2% for only 4 instances. Also, we can conclude that proposed VNS obtains the best improvements especially on very large instances ($T \geq 15$, $n \geq 20$ and $m \geq 5$). Notably, solutions for instances “r125.1cdy.col” and “r125.5dy.col” with respective parameters $T$, $n$, and $m$ equal to (75, 125, 23) and (38, 125, 18), improved for 37.1% and 32.4%. Similarly, two instances of “zeroin” type with values parameters $T$, $n$, and $m$ up to (35, 211, 15) have the improvement of 10% and higher.

In Table 8 average objective function values (Avg.), average running time in seconds (Time(s)), average percentage deviation with respect to the best solution found (Dev(%)), and the number of times only that method matches the best result (#Best), are presented. Average value and average time for IM, CPA, GRASP, GRASP+EC are taken from [11]. Average value and average time for BVNS, RSVNS, and proposed VNS, together with the average deviation from the best known result, shown in the second column of Tables 6 and 7, are calculated in accordance with the data presented in Tables 6 and 7. The number of times a method matches the best result is calculated as the number of instances for which the best result is matched only by using that method. For instance, the best result for “mpeg2end2dy” and “mug100_1dy” was reached only by IM. In analogy to that, for 2 instances the best results were obtained only by the BVNS, for 9 instances only by the RSVNS, and for 10 instances the best results were obtained only by the
proposed VNS. Given the fact that testings of considered methods were preformed on computers of different specifications, in order to compare running times of the proposed VNS algorithm with running times of the other considered algorithms, specifically, in order to compare the running times of the proposed VNS algorithm with the running times of the BVNS and RSVNS methods, a certain scaling factor should be applied. This scaling factor can be calculated as ratio between CPU mark (from https://www.cpubenchmark.net/compare.php) of computers performances. Since CPU of the computer used for testings of BVNS and RSVNS was marked as 4944 and CPU of ours as 1719, we can conclude that their computer is almost 3 times faster than ours. Regardless of the fact that average execution time of the proposed VNS algorithm is longer than the running time of other considered methods, longer execution time can be attributed to instances for which better objective function value is obtained. Now, as it can be seen from Table 8, although average value of the tested instances got using the proposed VNS method is the second best, the average deviation from the best known solution value obtained by the proposed VNS algorithm is the best. Indeed, solution value of the proposed VNS algorithm differ from the average value of the best known solutions in less than 1%. Further more, by using the proposed VNS algorithm for the biggest number of instances, the best known solution value is introduced.

Table 8: Comparisons between methods considered in this paper

|      | Avg. Time (s) | Dev (%) | #Best |
|------|--------------|---------|-------|
| IM   | 1,378,609.24 | 300.72  | 76.81 | 2    |
| CPA  | 1,375,151.61 | 300.48  | 477.73| 0    |
| GRASP| 1,110,887.76 | 300.45  | 29.65 | 0    |
| BVNS | 1,006,597.74 | 130.55  | 13.94 | 0    |
| RSVNS| 1,003,751.30 | 35.62   | 1.97  | 9    |
| VNS  | 1,004,086.27 | 539.78  | 0.85  | 10   |

In order to confirm the differences among the presented algorithm and algorithms known from the literature, we performed the Friedman non-parametric statistical test with all the individual values. The Friedman test ranks each algorithm in all instances according to the quality of the solution obtained, giving rank 1 to the best algorithm, 2 to the second one, and so on. If the averages differ greatly, the associated p-value or significance will be small. Similarly to the test presented in [16], the Friedman non-parametric statistical test is preformed for IM, GRASP+EC, BVNS, RSVNS, and VNS. The resulting p-value (lower than 0.001) obtained in this experiment clearly indicates that there are statistically significant differences among the tested methods. More precisely, the average rank values produced by this test are 3.9886 (IM), 4.0682 (GRASP+EC), 2.9091 (BVNS), 2.0227 (RSVNS), and 2.0114 (VNS), but additional p information shows that VNS and RSVNS statistically significantly differ from IM, GRASP, GRASP+EC, and BVNS but not from each other.

We conducted an additional statistical test (Wilcoxon signed rank test) to perform pair-wise comparisons between our method (VNS) and the previous al-
A New VNS approach

6. CONCLUSIONS

This paper presents our improvement of the existing ILP formulation for DMAP. The proposed cost function covers all oversights we referred to and therefore, with the proposed improvements, our new ILP formulation corresponds to the DMAP completely. In addition, we presented a new ILP formulation and a new metaheuristic approach based on the Variable Neighborhood Search. In order to compare the results of the proposed VNS heuristic and the methods previously established, we tested all of them on the same set of instances. From the computational results we can conclude that the proposed VNS heuristic effectively solves DMAP, providing (including CPLEX solutions of the new ILP formulation) 20 new best known solution values. Results obtained by the VNS are better than the results obtained by each method considered individually and better or equal to the results obtained by all methods previously established. Indeed, results obtained by the VNS are better or equal to the best result obtained by IM, CPA, GRASP, GRASP+EC, BVNS and RSVNS, combined for 25 instances. Execution time could not be easily compared given that almost all of the proposed methods for solving DMAP were tested on different types of computers, but all solutions were reached in a reasonable time limit or even less than 1,800 seconds.

Building similar dynamic memory allocation problems in embedded systems such as problems which involve calculation memory, possible movings during time blocks, conflicts which involve three or more data structures at the same time could be considered as the future work. Also, implementation of parallelization into VNS heuristic can be considered as well.

Acknowledgments: The authors are grateful to anonymous referees for their remarks and suggestions that helped to improve the manuscript. This research was supported by the Research Grants 174010 and TR36006 of the Serbia Ministry of Education, Science and Technological Developments.
REFERENCES

[1] Atienza, D., Mamagkakis, S., Poletti, F, Mendias, J. M., Catthoor, F., Benini, L., Soudris, D., “Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems”, INTEGRATION, the VLSI journal, 39 (2) (2006) 113–130.

[2] Hanafi, S., Lazić, J., Mladenović, N., Wilbaut, C., Crévits, L., “New variable neighbourhood search based 0-1 MIP heuristics”, Yugoslav Journal of Operations Research, 25 (3) (2015) 343-360.

[3] Hansen, P., Mladenović, N., “An introduction to variable neighborhood search”, Meta-heuristics, Springer, Boston, MA, 1999, 433-458.

[4] Hansen, P, Mladenović, N., Pérez, J.A.M., “Variable neighbourhood search: methods and applications”, Annals of Operations Research, 175 (1) (2010) 367-407.

[5] Hansen, P., Mladenović, N., “Variable neighborhood search: Principles and applications”, European journal of operational research, 130 (3) (2001) 449–467.

[6] Hansen, P., Mladenović, N., Developments of variable neighborhood search, Essays and surveys in metaheuristics, Springer, 2002, 415–439.

[7] Hansen, P., Mladenović, N., Variable neighbourhood search, Handbook of Metaheuristics, Springer, Boston, MA, 2003, 145-184.

[8] Ivanović, M., Dugošija, D., Savić, A., Urošević, D., “A new integer linear formulation for a memory allocation problem”, in proc. Balcor 2013, XI Balcan Conference on Operational Research, 2013, 284 288.

[9] Mladenović, N., Hansen, P., “Variable neighborhood search”, Computers & Operations Research, 24 (11) (1997) 1097–1100.

[10] Mladenović, N., “A variable neighborhood algorithm - a new metaheuristic for combinatorial optimization”, Optimization Weeks, 112 (1995) 112-112.

[11] Sevaux, M., Rossi, A., Soto, M., Duarte, A., Martí, R., “Grasp with ejection chains for the dynamic memory allocation in embedded systems”, Soft Computing, 18 (8) (2014) 1515–1527.

[12] Soto, M., Rossi, A., Sevaux, M., “A mathematical model and a metaheuristic approach for a memory allocation problem”, Journal of Heuristics, 18 (1) (2012) 149–167.

[13] Soto, M., Rossi, A., Sevaux, M., “Two iterative metaheuristic approaches to dynamic memory allocation for embedded systems”, European Conference on Evolutionary Computation in Combinatorial Optimization, Springer, Berlin, Heidelberg, 2011, 250–261.

[14] Soto, M., Sevaux, M., Rossi, A., Laurent, J., Memory Allocation Problems in Embedded Systems: Optimization Methods, John Wiley & Sons, Inc, Hoboken, NJ, 2013.

[15] Sánchez-Oro, J., Sevaux, M., Rosi, A., Martí, R., Duarte, A., “Solving dynamic memory allocation problems in embedded systems with parallel variable neighborhood search strategies”, Electronic Notes in Discrete Mathematics, 47 (2015) 85–92.

[16] Sánchez-Oro, J., Sevaux, M., Rosi, A., Martí, R., Duarte, A., “Improving the performance of embedded systems with variable neighborhood search”, Applied Soft Computing, Elsevier, 53 (2017) 217-226.

[17] Wuytack, S., Catthoor, F, Nachtergaele, L., De Man, H., “Power exploration for data dominated video applications”, in: Proceedings of the 1996 international symposium on Low power electronics and design, IEEE Press, 1996, 359–364.