A Binary Algebraic Differential Evolution for the MultiDimensional Two-Way Number Partitioning Problem

Valentino Santucci[1], Marco Baioletti[2], Gabriele Di Bari[2], and Alfredo Milani[2]

[1] Department of Humanities and Social Sciences
University for Foreigners of Perugia, Italy
valentino.santucci@unistrapg.it

[2] Department of Mathematics and Computer Science
University of Perugia, Italy
{marco.baioletti,alfredo.milani}@unipg.it, gabriele.dibari@unifi.it

Abstract. This paper introduces MADEB, a Memetic Algebraic Differential Evolution algorithm for the Binary search space. MADEB has been applied to the Multidimensional Two-Way Number Partitioning Problem (MDTWNPP) and its main components are the binary differential mutation operator and a variable neighborhood descent procedure. The binary differential mutation is a concrete application of the abstract algebraic framework for the binary search space. The variable neighborhood descent is a local search procedure specifically designed for MDTWNPP. Experiments have been held on a widely accepted benchmark suite and MADEB is experimentally compared with respect to the current state-of-the-art algorithms for MDTWNPP. The experimental results clearly show that MADEB is the new state-of-the-art algorithm in the problem here investigated.

Keywords: Binary Algebraic Differential Evolution · Multidimensional Two-Way Number Partitioning Problem · Variable Neighborhood Descent.

1 Introduction

In this paper we propose a memetic binary variant of the Algebraic Differential Evolution (ADE) algorithm [1], namely, MADEB, for the multidimensional two-way number partitioning problem (MDTWNPP) [2].

MDTWNPP has been originally introduced in [2] as a multidimensional generalization of the more famous number partitioning problem (NPP) [3].

An instance of MDTWNPP is given as a set S of n real-valued vectors of dimension d, i.e., $S = \{w_k \in \mathbb{R}^d : 1 \leq k \leq n\}$. The objective is to partition S into two subsets S_0 and S_1, i.e., $S_0 \cup S_1 = S$, $S_0 \cap S_1 = \emptyset$, and such that the sum of the vectors in S_0 is as close as possible to that of S_1. Formally, given the

partition $\{S_0, S_1\}$, the objective function to be minimized is

$$f(S_0, S_1) = \max_{1 \leq j \leq d} \left| \sum_{w_i \in S_0} w_{ij} - \sum_{w_i \in S_1} w_{ij} \right|, \tag{1}$$

where w_{ij} is the j-th component of the vector w_i.

Clearly, any partition $\{S_0, S_1\}$ can be represented as a n-length bit-string x, i.e., $x \in \mathbb{B}^n$, in such a way that $x(i) = 0$ when $w_i \in S_0$, and $x(i) = 1$ when $w_i \in S_1$. Therefore, MDTWNPP is a binary optimization problem.

MDTWNPP is NP-hard because, for $d = 1$, it reduces to the optimization variant of classical NPP. Moreover, as described in [2], MDTWNPP is even more difficult than NPP, because the computational complexity of its instances does not decrease together with the ratio between the number of bits needed to express any input element and the quantity of input elements (as it has been observed to happen in NPP instances [3, 4]). This greater complexity has been also experimentally confirmed in [5], where it has been observed that CPLEX, applied to an integer linear programming model for MDTWNPP, has never been able to compute a better lower bound than the trivial one, i.e., zero.

For these reasons, meta-heuristic algorithms have recently been applied to MDTWNPP [6, 7, 5]. Following this line of research, here we introduce a binary variant of the Algebraic Differential Evolution (ADE) algorithm specifically designed for MDTWNPP. ADE has been originally introduced in [1, 8] and, though it has been extensively applied to permutation-based optimization problems [9–13], the abstract algebraic framework at its bases works with any finitely generated group [1, 14].

In this paper, we show how it is possible to see the binary search space $\mathbb{B}^n$ as a finitely generated group, thus we formally derive an algebraically and geometrically consistent implementation of the differential mutation operator for the $\mathbb{B}^n$ space. It is worthwhile to note that our proposal is substantially different from the other binary DE schemes in the literature. For example, [15] and [16] propose two binary DEs that mainly work in the continuous space and decode back the numerical vectors to bit-strings as soon as they are needed.

Besides, we introduce two new algorithmic components: (i) a modified initialization scheme, (ii) an effective variable neighborhood descent scheme for MDTWNPP.

Due to these considerations, we will call our algorithm MADEB (Memetic Algebraic Differential Evolution for the Binary space) in the rest of paper, that has been organized as follows. Section 2 provides a brief review of the state-of-the-art algorithms for MDTWNPP. The main scheme of MADEB is introduced in Section 3. Its key operators, i.e., the binary algebraic differential mutation and the variable neighborhood descent are introduced and motivated in Sections 4 and 5. Computational experiments have been described and analyzed in Section 6. Finally, conclusion are drawn in Section 7 which also depicts some possible future lines of research.

2 Related Work

In this section we describe the main approaches used to solve MDTWNPP. Although this problem is a direct generalization of the number partitioning problem [3], many techniques used to solve the latter cannot be extended to the multidimensional case, like, for instance, the Karmarkar-Karp heuristic (see [2]).

The MDTWNPP has been defined in [2], where an integer linear programming formulation is provided. Here, a set of 210 benchmark instances have been randomly generated and solved by using the linear programming solver CPLEX.

A memetic algorithm (MA) has been defined in [6] for solving a generalization of the MDTWNPP, called multidimensional multiway number partitioning problem, in which the vectors must be partitioned in $p \geq 2$ subsets. The MA is a genetic algorithm which produces, at each generation, 10 offsprings for every population individual by means of crossover, mutation and local search operators. The best offsprings are then selected for the next-generation population. The local search method uses a k-change neighborhood, for $k = 1, 2, 3$, where k denotes the number of bits changed by a single move. The authors performed computational experiments also in the case of $p = 2$, i.e., in the MDTWNPP problem. The results show that MA outperforms CPLEX in most instances.

Two other meta-heuristic approaches have been introduced in [7].

The first one is a VNS-like procedure which operates on a incumbent solution x, represented as a bit-string. A series of increasing neighborhoods $N_k(x)$ are employed in the shaking phase, along with a local search whose elementary step is to flip a 0-bit and a 1-bit of x. This corresponds to swap two vectors: one vector moves from partition S_0 to partition S_1, while the other one moves in the opposite direction. The generic neighborhood $N_k(x)$ is defined as the set of all the bit-strings having Hamming distance k from x. The parameter k is increased from $k_{min} = 2$ to $k_{\max} = \min\{30, \lfloor n/4 \rfloor\}$ (in a circular way) at every iteration where the new solution does not improve the incumbent one.

The second meta-heuristic uses an Electromagnetism-like (EM) approach. Solutions are represented as real vectors with components in $[0, 1]$. The partition associated to a solution is defined by applying a threshold equal to 0.5. At each generation, every solution undergoes to a local search and scale operators, then all the solutions are moved according to "electromagnetic forces" that can be attractive or repulsive and depend on the objective values.

The experimental results show that VNS and EM obtained comparable performances and both outperform MA and CPLEX.

Finally, in [5], a GRASP equipped with an Exterior Path Relinking method is described. The algorithm evolves a set of solutions, called elite set. At each step, the GRASP procedure produces a new solution by means of two operations: construction and local improvement. The former operation builds-up a solution by means of a greedy method, while the latter iteratively improves the incumbent solution by using a possibly restricted local search in the space of the 2-change neighborhood. Then, the Path Relinking phase explores a path from the new solution s_i to a randomly selected solution s_G in the elite set (Interior PR) or beyond s_G (Exterior PR), returning the best solution found in the path. The

configuration with the Exterior Path Relinking, i.e., GRASP+ePR, resulted in better performances and, compared with VNS and CPLEX, it almost always outperformed them, thus representing the current state-of-the-art algorithm for MDTWNPP.

3 The general scheme of MADEB

The classical Differential Evolution (DE) [17, 18] is a popular and effective evolutionary algorithm for continuous optimization problems that iteratively evolves a population of numerical vectors by means of three genetic operators: differential mutation, crossover, and selection. In particular, the differential mutation is widely considered to be the key component of DE [19]. Despite its proven effectiveness in numerical optimization [20, 21], most of the DE schemes for combinatorial problems are not so effective, perhaps, because their search behavior is loosely connected to the working mechanisms of continuous DE (see, for example, [15, 16]). In order to fill this gap, in [1], an original algebraic framework has been introduced in order to design a differential mutation for combinatorial search spaces in such a way that it consistently simulates the behavior of its numerical counterpart. The framework abstractly defines the design of the combinatorial differential mutation in any finitely generated group. In the previous works, implementations for the permutations search space have been proposed [8, 12]. In MADEB, we introduce the first instantiation of the abstract combinatorial differential mutation for binary search spaces.

Therefore, MADEB evolves a population of N bit-strings by iterative applications of the following operators: binary algebraic differential mutation, variable neighborhood descent, and selection. Its general scheme is depicted in Algorithm 1.

Algorithm 1 General scheme of MADEB

```
1: function MADEB(N, init)
2:     Initialize N bit-strings {x_1, ..., x_N} by means of the init procedure
3:     while termination condition has not been satisfied do
4:         for i = 1 to N do
5:             y_i ← AlgebraicDifferentialMutation(x_i)
6:             z_i ← VariableNeighborhoodDescent(y_i)
7:         end for
8:         for i = 1 to N do
9:             x_i ← Selection(x_i, z_i)
10:        end for
11:        if x_best has not been updated in the last 1 000 generations then
12:            Reinitialize the bit-strings in {x_1, ..., x_N} \ {x_best} by using init
13:        end if
14:    end while
15:    return the best visited bit-string x_best
16: end function
```

Two population initialization procedures are considered. When $init = R$, the bit-strings are randomly initialized as usual, i.e., every bit $x_i(j)$, with $1 \leq i \leq N$ and $1 \leq j \leq n$, is initialized to 0 or 1, with probability 0.5. When $init = U$, a probability value $p_i \in [0, 1]$ is randomly generated for every individual x_i, and its bits $x_i(j)$ are independently set to 1 with probability p_i, or 0 otherwise. In this way, the expected number of 1-bits throughout the population individuals is uniformly distributed. The rationale behind the $init = U$ scheme is to allow a more diverse population initialization.

For every population individual x_i, *AlgebraicDifferentialMutation* generates a mutant y_i as follows:

$$y_i \leftarrow x_i \oplus F_i \odot (x_{r_1} \ominus x_{r_2}), \tag{2}$$

where: $F_i > 0$ is the DE scale factor, $r_1, r_2 \in [1, N]$ are randomly generated indexes different between them and with respect to i, and the $\oplus, \ominus, \odot$ are the binary versions of our algebraic operators that are introduced and discussed in Section 4. It is worthwhile to note that the scale factor is self-adapted during the evolution by means of the the popular jDE scheme [22]. In particular: every individual has its own F_i value; the differential mutation is computed using a scale factor randomly generated in $[0.1, 2]$ with probability 0.1, or F_i otherwise; if the trial individual replaces x_i during the selection, then F_i is updated with the employed scale factor.

After the differential mutation, the mutant y_i undergoes a local search phase. The local search procedure *VariableNeighborhoodDescent* adopts two neighborhood definitions and generates the trial individual z_i in such a way that z_i is a local minimum of both neighborhoods. *VariableNeighborhoodDescent* is further described in Section 5.

The *Selection* procedure replaces x_i with z_i if and only if $f(z_i) < f(x_i)$, where f is the fitness function defined in equation (1). Moreover, if the best population individual x_{best} has not been updated during the last 1 000 generations, the population, except x_{best}, is randomly reinitialized by means of the chosen *init* procedure.

Summarizing, MADEB requires only two parameters to be set: the population size N, and the initialization procedure $init \in \{R, U\}$.

4 Algebraic Differential Mutation for the Binary Space

The binary algebraic differential mutation is the main component of MADEB. In the following, after briefly recalling the abstract algebraic framework (originally introduced in [1, 12]), we introduce its implementation for the binary space. This allows to present the binary algebraic differential mutation, that is further analyzed in the last part of this section.

4.1 Abstract Algebraic Framework

A combinatorial search space is a set X of discrete solutions. X forms a group if there exists a binary operation $\star$ between the elements of X such that: $\star$ is

associative, there exists a unique neutral element, and any $x \in X$ has a unique inverse $x^{-1} \in X$. The group is finitely generated if there exists a subset $H \subseteq X$ such that every $x \in X$ can be decomposed as $x = h_1 \star h_2 \star \cdots \star h_l$, for some $h_1, h_2, \ldots, h_l \in H$. H is the *generating set* of X and its elements are called *generators*.

A search space representable by a finitely generated group $(X, \star, H)$ can be visualized by its Cayley graph $\mathcal{C}(X, \star, H)$, whose vertices are the solutions in X, and there is an arc from $x \in X$ to $y \in X$, labeled by $h \in H$, if and only if $x \star h = y$.

Interestingly, the labels in a shortest path from x to y corresponds to the sequence of generators in a minimal decomposition of $x^{-1} \star y$. We denote the length of the minimal decomposition (or, equivalently, of the shortest path) by $|y^{-1} \star x|$.

These aspects allow us to define the algebraic operators $\oplus, \ominus, \odot$ that simulate, in the Cayley graph (i.e., the combinatorial search space), the geometric behavior of their numerical counterparts.

We define $x \oplus y := x \star y$ and $y \ominus x := x^{-1} \star y$. The sum $s = x \oplus y$ can be interpreted in the Cayley graph as starting from vertex x and moving (towards s) by iteratively choosing the arcs corresponding to a minimal decomposition of y. The difference $d = y \ominus x$ can be interpreted as the composition of the generators in a shortest path going from x to y, thus d is a synthetic representation of this path. As in the numerical case, $x \oplus (y \ominus x) = x \star (x^{-1} \star y) = y$.

Moreover, given a scalar $F \geq 0$, the multiplication $F \odot x$ can be described as follows. Let us interpret x as the representation of a shortest path between two arbitrary vertices of the Cayley graph, then, when $F \leq 1$ (or $F > 1$), $F \odot x$ represents, a truncation (or an extension) of the path represented by x. Algebraically, this geometric interpretation can be encoded by requiring $z = F \odot x$ to satisfy these properties: (i) $|z| = \lceil F \cdot |x| \rceil$, (ii) if $F \leq 1$, the sequence of generators in a minimal decomposition of z is a prefix of the sequence of generators in a minimal decomposition of x, or (iii) vice versa, when $F > 1$. Since, in general, minimal decompositions (thus shortest paths) are not unique, there can be multiple $z \in X$ satisfying these properties, therefore, we define $\odot$ as a stochastic operator that randomly returns an element with the given properties.

Summarizing, the geometric interpretation of the algebraic differential mutation of equation (2) is as follows. Let first compute the generators in a (shortest) path from x_{r_2} to x_{r_1}; truncate (or extend) the sequence of generators considering the scalar F; start from the vertex x_i and move, towards the result y_i, by following one by one the arcs labeled with the generators in the truncated (or extended) sequence.

4.2 Binary Algebraic Differential Mutation

$\mathbb{B}^n$ is the set of all the bit-strings of length n and it forms an Abelian group with respect to the bit-wise XOR operation $\veebar$. Indeed, $\veebar$ is commutative and

associative, the zero bit-string $\mathbf{0}$ is the neutral element, and the inverse of each $x \in \mathbb{B}^n$ is itself, i.e., $x = x^{-1}$.

Given $x \in \mathbb{B}^n$, we denote by $x(i)$ the i–th bit of x, for $1 \leq i \leq n$. We recall that the Hamming weight $|x|$ is the number of 1-bits of x. The Hamming distance between two bit-strings x and y is the number of positions i such that $x(i) \neq y(i)$.

Moreover, $\mathbb{B}^n$ is a finitely generated group. Its generating set U is composed by the n bit-strings with Hamming weight equal to 1, thus the generic generator $u_i \in U$, for $1 \leq i \leq n$, is such that $u_i(i) = 1$, while the rest of its bits are 0.

Therefore, any $x \in \mathbb{B}^n$ can be written as $x = u_{i_1} \veebar u_{i_2} \veebar \ldots \veebar u_{i_l}$, where $i_1, i_2, \ldots, i_l$ are the indexes of the 1-bits of x. Clearly, $l = |x|$. The decomposition is minimal and unique, up to reordering the indexes $i_1, i_2, \ldots, i_l$. We exploit this property and we represent the minimal decomposition of $x \in \mathbb{B}^n$ as the set $U_x = \{u_i \in U : x(i) = 1\}$. Note anyway that any ordering of the generators in U_x is a sequence that fulfills the abstract framework definitions.

Importantly, note that, for each $x \in \mathbb{B}^n$, the application of the generator u_i to x, i.e., $x \veebar u_i$, corresponds to flipping the i–th bit of x.

As any other finitely generated group, $(\mathbb{B}^n, \veebar, U)$ has its associated Cayley graph. Since $\veebar$ is commutative and each bit-string is the inverse of itself, the Cayley graph $\mathcal{C}(\mathbb{B}^n, \veebar, U)$ reduces to an indirected labelled graph, which corresponds to the hypercube with n vertices, where all the pairs of bit-strings, differing in a single bit i, are connected by an edge labelled with u_i.

Therefore, by following the abstract definitions given in Section 4.1, it is now possible to concretely derive the operations $\oplus, \ominus, \odot$ for the binary space.

The addition $\oplus$ is defined as $x \oplus y := x \veebar y$ for $x, y \in \mathbb{B}^n$. The subtraction uses the property that $x^{-1} = x$ for each $x \in \mathbb{B}^n$, hence $y \ominus x := x \veebar y$. Note that, in this particular group, $\oplus$ and $\ominus$ coincide.

Given $F \geq 0$ and $x \in \mathbb{B}^n$, the stochastic scalar multiplication $F \odot x$ is defined as randomly selecting a $z \in \mathbb{B}^n$ such that its decomposition U_z: (i) has size $k = \lceil F \cdot |x| \rceil$, and (ii) when $F \leq 1$, $U_z \subseteq U_x$, while, (iii) if $F > 1$, $U_z \supseteq U_x$. It is easy to see that any ordering of the generators in U_z satisfies the three properties described in Section 4.1.

When $F \leq 1$, z is computed by randomly selecting one of the $\binom{|x|}{k}$ subsets of size k of U_x.

When $F > 1$, U_z is computed as $U_x \cup A$, where A is randomly selected among the $\binom{n-|x|}{k-|x|}$ subsets of $U \setminus U_x$. This is equivalent to randomly flipping some of the 0-bits of x.[3]

Since this is a concrete realization of the abstract framework of Section 4.1, all the geometric considerations previously provided are valid also in this search space. In particular, it is interesting to note that the elementary search moves here considered are bit-flip moves, that is, the most common and natural moves adopted in the binary search space.

[3] For this reason, $|F \odot x|$ cannot be larger than n, thus we truncate F to $F^{(x)}_{\max} = \frac{n}{|x|}$ whenever $F > F^{(x)}_{\max}$.

4.3 Search characteristics of the Binary Differential Mutation

Here we analyze the binary implementation of the algebraic differential mutation provided in equation (2).

We start by describing the computation of the mutant $y_i \leftarrow x_i \oplus F \odot (x_{r_1} \ominus x_{r_2})$ according to the definitions introduced in Section 4.2. Lets first compute the bit-wise XOR between x_{r_1} and x_{r_2}. This corresponds to selecting the positions where the bits of x_{r_1} and x_{r_2} differ. A subset (or a superset) of these positions is computed by considering the scalar F. Finally, x_i is moved, towards the result y_i, by flipping all the bits of x_i at the positions previously computed.

As an illustrative example, let consider $x_i = (1010)$, $x_{r_1} = (1001)$, $x_{r_2} = (1110)$, and $F = 0.66$. The difference between x_{r_1} and x_{r_2} is computed as $d = x_{r_1} \ominus x_{r_2} = x_{r_1} \veebar x_{r_2} = (0111)$. Its Hamming weight is $|d| = 3$, thus we have to randomly select $\lceil F \cdot |d| \rceil = 2$ 1-bits from d. Let's choose the last two 1-bits of d, thus $F \odot d = (0011)$. Finally, $y_i = x_i \oplus (F \odot d) = x_i \veebar (F \odot d) = (1001)$.

It is interesting to note that, by denoting the Hamming distance with d_H, we have that $d_H(x_i, y_i) = F \cdot d_H(x_{r_1}, x_{r_2})$, i.e., the amount of perturbation applied to x_i is decided by the scale factor F and the two randomly selected population individuals x_{r_1} and x_{r_2}. Moreover, even the positions where the bit-flips are applied are computed by means of x_{r_1} and x_{r_2}. This behavior is quite analogous to what happens when classical differential mutation is applied in the continuous space [17].

However, the structural characteristics of the binary space can introduce some issues in the search behavior of the binary differential mutation. Indeed, binary $\oplus$ and $\ominus$ are actually the bit-wise XOR operator. Moreover, the bit-strings x_i, x_{r_1}, x_{r_2}, involved in the differential mutation formula, are three individuals taken from the current MADEB population. This implies that, when the population reaches consensus on a bit (i.e., when all individuals have their i-th bit set to the same value), it is impossible to flip that bit in the future generations by only using the binary differential mutation with $F \leq 1$. This is the reason of why we use an interval with a right bound larger than 1 for the scale factor F, i.e., $F \in [0.1, 2]$.

Furthermore, it is worthwhile to note that most of the binary crossovers in the literature are somehow special cases of our binary differential mutation. Indeed, a binary crossover between two generic $x, y \in \mathbb{B}^n$ usually computes an offspring $z \in \mathbb{B}^n$ such that the bit value $z(j)$ is equal to $x(j)$ or $y(j)$, for $1 \leq j \leq n$. Possible examples are the uniform crossover, the one-point crossover or the more general k-points crossover [23]. It is easy to see that the computation of such an offspring can be reproduced in the algebraic framework as $z = x \oplus F \odot (y \ominus x)$, by using $F \in [0, 1]$. For instance, the uniform crossover which takes $z(j) = x(j)$ with probability 0.5, or $z(j) = y(j)$ otherwise, for $j = 1, \ldots, n$, is equivalent on average to $z = x \oplus 0.5 \odot (y \ominus x)$. Other crossover operators can be obtained by simply considering different selection strategies (other than the random one) in the $\odot$ definition. Therefore, since a binary crossover is a special case of our binary differential mutation, we have decided to not employ the crossover operator in MADEB.

5 Variable Neighborhood Descent for MDTWNPP

In MADEB, every mutant y undergoes a local search procedure implemented as a variable neighborhood descent scheme.

The *VariableNeighborhoodDescent* procedure uses two different neighborhoods N_1 and N_2. $N_1(y)$ is the classical bit-flip neighborhood, while $N_2(y)$ is the 2-change neighborhood that contains all the bit-strings which can be obtained from y by flipping one 0-bit and one 1-bit.

The two neighborhoods are explored alternatively until no better solution is found as depicted in Algorithm 2.

Algorithm 2 Pseudocode of *VariableNeighborhoodDescent*

```
1: function VariableNeighborhoodDescent(y)
2:     imp1 ← true
3:     while imp1 do
4:         z ← y
5:         /* Local Search in N_1 */
6:         imp2 ← true
7:         while imp2 do
8:             Randomly permute the solutions in N_1(y)
9:             y' ← the first solution in N_1(y) s.t. f(y') < f(y), or nil if no improvement
10:            if y' is not nil then
11:                y ← y'
12:            else
13:                imp2 ← false
14:            end if
15:        end while
16:        /* Local Search in N_2 */
17:        imp2 ← true
18:        while imp2 do
19:            Randomly permute the solutions in N_2(y)
20:            y' ← the first solution in N_2(y) s.t. f(y') < f(y), or nil if no improvement
21:            if y' is not nil then
22:                y ← y'
23:            else
24:                imp2 ← false
25:            end if
26:        end while
27:        /* Main loop operations */
28:        if f(y) = f(z) then
29:            imp1 ← false
30:        end if
31:    end while
32:    return y
33: end function
```

A first-improvement style is used in every iteration of the neighborhood local searches (lines 4–8 and 9–13 of the pseudocode), i.e., as soon as an improving neighbor is found, it is returned. Moreover, the neighbors in N_1 and N_2 are randomly scanned. Finally, it is important to note that the computation of a neighbor solution is not made by scratch, but incrementally with respect to the incumbent solution. Indeed, by keeping track of the two partition sums, it is possible to compute the objective value of a neighbor in both N_1 and N_2 in $\Theta(d)$ time.

It is easy to prove that the overall computational cost for each iteration of the local search in N_1 is $\Theta(n \cdot d)$, while for N_2 is $\Theta(n^2 \cdot d)$.

6 Experiments

The behavior of MADEB in the MDTWNP problem has been experimentally analyzed by considering the benchmark set of 210 instances provided in [2], where the author proposed 5 instances for each problem configuration n, d with $n \in \{50, 100, 200, 300, 400, 500\}$ and $d \in \{2, 3, 4, 5, 10, 15, 20\}$.

Following the methodology adopted in [5], we have divided the instances in two sets: tuning and test instances. The tuning set consists of the first two instances in every problem configuration n, d and it has been adopted in order to select a good setting for MADEB. The rest of the instances have been devoted to experimentally compare the tuned MADEB with the state-of-the-art algorithms for MDTWNPP.

In order to make a fair comparison, as in [5], for each problem instance, MADEB has been executed 10 times with a computational budget of 600 seconds per execution. The experiments were carried out on a machine equipped with an Intel Xeon X5650 processor clocking at 2.67 GHz, which has a very similar computational power with respect to the machine adopted in [5].

In the following, we present the tuning of the MADEB setting and the experimental comparison with the state-of-the-art results for MDTWNPP.

6.1 Experimental Tuning of MADEB

MADEP has two parameters to be set: the population size N, and the initialization procedure *init*. Eight MADEB settings have been analyzed, with $N \in \{50, 100, 150, 200\}$ and $init \in \{R, U\}$, by performing a full factorial analysis on 84 tuning instances.

The average objective values obtained by every MADEB setting, on its executions on every tuning instance, have been recorded. In Table 1, we provide the ranks of the MADEB settings averaged over all the tuning instances.

The best setting is $(N = 50, init = U)$, which reached the lowest average rank of 4.04. Therefore, this is the setting used for the experimental comparison discussed in Section 6.2.

Moreover, as recommended in [24], we have conducted a non-parametric Friedman test in order to analyze the statistical differences among the settings.

Table 1: Average ranks of MADEB settings in the experimental tuning.

MADEB setting	**Avg Rank**
$N = 50, \text{init} = U$	4.04
$N = 50, \text{init} = R$	4.27
$N = 150, \text{init} = U$	4.40
$N = 100, \text{init} = U$	4.43
$N = 200, \text{init} = R$	4.43
$N = 100, \text{init} = R$	4.45
$N = 200, \text{init} = U$	4.56
$N = 150, \text{init} = R$	4.95

Since the Friedman test shows that there is no significant difference among the eight settings, MADEB can be considered a robust algorithm.

6.2 Comparison with state-of-the-art MDTWNPP algorithms

In this section we compare MADEB, using the setting ($N = 50, init = U$), with the state-of-the-art MDTWNPP algorithms described in Section 2.

As in [5], MADEB has been executed 10 times on every one of the 126 test instances. Its results have been compared with those obtained by GRASP+ePR [5], VNS [7], and CPLEX [2]. The results for the competitor algorithms have been directly obtained from the supplementary material of [5] available online at https://sci2s.ugr.es/MDTWNP.

The performance of each algorithm A, on every instance i, is measured by the commonly adopted average relative percentage deviation (ARPD) [25]:

$$ARPD_i^A = \frac{1}{k} \sum_{j=1}^{k} \frac{(A_i^j - Best_i)}{Best_i} \times 100, \tag{3}$$

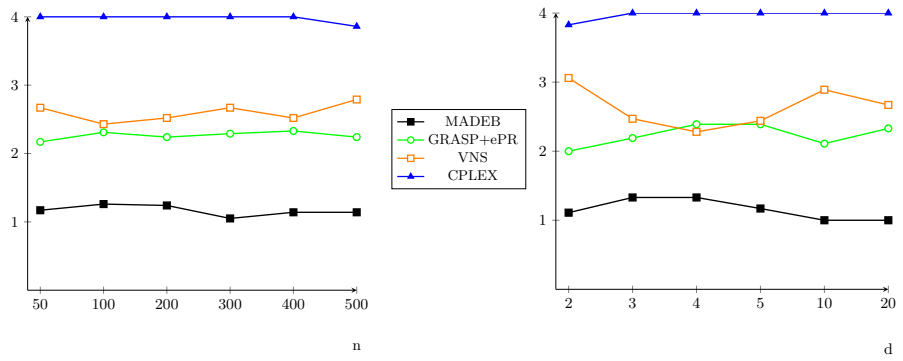
where A_i^j is the objective value obtained by the algorithm A in its j-th run on the instance i, and $Best_i$ is the best objective value achieved among all executions of all the algorithms on the problem instance i.

Moreover, in order to detect the statistical differences among the ARPD results, as suggested in [24], the non-parametric Friedman test and the Finner post-hoc procedure have been applied.

For each problem instance, the best objective value and the ARPDs of each algorithm are provided in Table 2. The best ARPD on each instance is reported in bold, while the best objective value is in bold when it has been reached by (at least) an execution of MADEB. The table is divided in six groups according to the different values of n. For any group of instances, the average rank of every algorithm is provided together with a symbol indicating the result of the statistical comparison with MADEB: "+" means that MADEB significantly outperforms the competing algorithm. Moreover, the average ranks aggregated for every value of n and d are shown, respectively, in Figures 1a and 1b.

Table 2: Experimental comparison of MADEB with state-of-the-art algorithms.

Problem Instance	Best Obj. Val.	MADEB	GRASP +ePR	VNS	CPLEX	Problem Instance	Best Obj. Val.	MADEB	GRASP +ePR	VNS	CPLEX
50_2c	**3.083**	**45.20**	76.86	141.71	1784492.93	300_2c	0.759	**198.30**	339.24	657.69	1529412.52
50_2d	**1.349**	113.36	**46.30**	306.27	3115763.60	300_2d	**0.433**	**272.93**	892.89	1378.71	863410.39
50_2e	**2.938**	**97.80**	119.91	189.55	1135570.52	300_2e	**0.917**	**33.82**	244.25	928.60	588013.41
50_3c	**142.935**	**1.07**	2.78	2.78	232764.59	300_3c	**214.729**	**4.69**	15.82	16.79	109847.42
50_3d	96.410	5.07	8.55	**4.13**	358105.58	300_3d	91.207	**26.42**	34.62	51.23	269626.01
50_3e	211.797	**1.59**	**1.59**	2.45	325728.98	300_3e	**167.205**	**10.95**	41.30	22.88	229009.18
50_4c	**452.007**	0.88	**0.81**	1.42	398.81	300_4c	**885.348**	**1.24**	1.52	12.57	127.67
50_4d	1018.573	**0.30**	0.42	0.42	192.14	300_4d	765.805	30.47	41.12	**27.66**	126.28
50_4e	1202.624	**0.30**	0.43	0.51	438.27	300_4e	736.131	**22.18**	23.63	22.56	135125.25
50_5c	1398.771	**0.39**	32.16	6.84	334.37	300_5c	**901.749**	**1.11**	1.32	126.17	430.23
50_5d	2269.686	**0.16**	0.17	0.47	63.77	300_5d	**2115.419**	**0.52**	19.29	7.01	178.64
50_5e	**4360.613**	**1.70**	4.35	2.83	40.99	300_5e	**1543.701**	**0.32**	0.57	25.15	177.60
50_10c	**14125.537**	**7.76**	23.15	26.68	71.35	300_10c	**15971.771**	**0.06**	0.13	5.23	68.21
50_10d	**14915.871**	**0.06**	3.38	23.47	50.64	300_10d	**15812.711**	**7.84**	17.66	10.35	59.91
50_10e	15356.800	**1.86**	7.75	8.64	96.02	300_10e	**15291.777**	**2.14**	10.39	12.64	142.85
50_15c	29456.850	**4.76**	18.34	24.96	78.96	300_15c	**22220.712**	**27.14**	42.70	42.66	138.53
50_15d	21655.890	**0.00**	28.88	59.13	133.79	300_15d	28891.210	**13.59**	18.29	20.76	77.21
50_15e	**31800.690**	**1.17**	5.86	14.54	47.81	300_15e	**28754.344**	**1.65**	4.61	20.00	81.33
50_20c	50560.860	**0.18**	9.32	6.64	34.47	300_20c	**34247.615**	**18.34**	41.50	48.49	77.57
50_20d	53955.960	**1.90**	4.62	6.46	33.10	300_20d	**42351.439**	**8.14**	20.75	21.77	67.56
50_20e	**48281.499**	**0.00**	21.09	9.86	47.95	300_20e	**37132.464**	**19.65**	32.14	34.36	108.58
Average Rank		**1.17**	2.17	2.67	4.00	**Average Rank**		**1.05**	2.29	2.67	4.00
Stat. Comp. vs MADEB			+	+	+	**Stat. Comp. vs MADEB**			+	+	+
100_2c	**0.768**	**307.97**	499.79	1129.65	2030759.38	400_2c	0.884	**123.39**	167.83	694.05	1419357.01
100_2d	**0.001**	**61140.00**	213560.00	643980.00	3833499900.00	400_2d	**1.164**	**87.21**	297.39	715.05	2524.57
100_2e	**1.067**	**14.55**	28.88	140.17	4919581.35	400_2e	**0.406**	**340.64**	636.31	1786.97	3609013.30
100_3c	231.512	**7.45**	15.14	7.58	330.18	400_3c	66.576	153.85	256.07	**139.45**	434203.65
100_3d	**243.989**	**1.20**	2.82	3.49	209597.57	400_3d	**212.073**	**4.11**	8.28	31.62	135968.24
100_3e	**135.305**	**4.51**	**4.51**	28.95	367653.59	400_3e	**155.001**	**10.63**	30.76	24.73	245907.45
100_4c	**896.038**	**0.65**	0.99	1.60	96.57	400_4c	807.262	**10.43**	11.19	25.30	218.84
100_4d	**1090.117**	0.67	**0.59**	1.57	44.96	400_4d	806.920	**1.25**	11.33	11.85	123275.06
100_4e	**508.289**	**1.32**	1.92	8.74	174.41	400_4e	725.451	**23.47**	28.36	26.62	116.37
100_5c	2769.736	1.93	2.21	**1.04**	63.89	400_5c	**902.131**	**1.27**	1.55	129.45	342.50
100_5d	**2969.833**	0.17	**0.16**	0.45	101.24	400_5d	1667.575	49.72	49.88	**21.07**	239.24
100_5e	2448.321	52.88	54.22	**30.54**	142.58	400_5e	**1595.032**	**0.77**	1.15	26.64	239.76
100_10c	11984.020	**15.69**	23.46	18.81	145.19	400_10c	14503.050	**7.97**	15.32	17.85	113.16
100_10d	14904.240	**4.35**	17.14	12.71	59.72	400_10d	**13723.390**	**5.42**	10.77	16.53	75.20
100_10e	12182.070	18.01	32.75	**12.60**	150.43	400_10e	11571.600	39.64	40.86	**35.61**	183.04
100_15c	**31112.332**	**3.23**	7.74	12.73	37.64	400_15c	**26815.632**	**3.60**	12.16	21.41	108.86
100_15d	**30690.906**	**0.47**	8.09	13.80	64.74	400_15d	26395.830	**16.89**	25.34	23.54	99.42
100_15e	**30250.569**	**1.35**	6.17	8.41	91.59	400_15e	**30105.307**	**1.85**	2.98	12.46	78.08
100_20c	**46021.607**	**7.91**	25.64	16.24	60.14	400_20c	**45374.378**	**2.07**	11.29	10.40	63.91
100_20d	**43786.661**	**7.53**	21.48	16.78	53.39	400_20d	**41384.465**	**4.87**	15.65	21.38	59.66
100_20e	**43923.294**	**10.32**	23.09	17.57	56.30	400_20e	39541.150	**9.05**	21.45	25.04	76.03
Average Rank		**1.26**	2.31	2.43	4.00	**Average Rank**		**1.14**	2.33	2.52	4.00
Stat. Comp. vs MADEB			+	+	+	**Stat. Comp. vs MADEB**			+	+	+
200_2c	0.690	**91.43**	241.59	436.93	6394972.46	500_2c	**0.578**	**64.71**	372.73	728.53	2333117.99
200_2d	0.237	**499.83**	1353.50	1846.75	11773317.72	500_2d	**0.202**	603.86	941.58	3457.08	**3.96**
200_2e	**0.451**	**138.14**	371.84	829.18	6777948.78	500_2e	0.205	**611.37**	1070.78	1679.32	7324290.24
200_3c	1.526	**159.95**	209.89	12237.23	36695640.50	500_3c	96.343	**39.49**	95.09	55.76	391556.89
200_3d	77.565	**46.30**	52.20	50.38	774080.36	500_3d	1.522	**330.93**	421.16	4294.56	20003973.59
200_3e	**105.284**	55.99	91.57	**54.65**	199011.93	500_3e	**3.056**	**84.79**	144.71	3267.62	9173885.60
200_4c	3.047	65.25	**50.26**	22036.19	65850.77	500_4c	**633.460**	**47.44**	69.00	85.70	103.02
200_4d	770.115	30.52	38.66	**28.90**	255.22	500_4d	**3.057**	**159.54**	194.40	23031.45	72864.34
200_4e	508.023	73.13	142.78	**43.84**	313.03	500_4e	552.412	**3.76**	53.09	44.45	153.71
200_5c	**3.052**	**79.97**	114.95	21439.79	163770.90	500_5c	**1891.494**	**23.11**	51.26	43.38	168.56
200_5d	2130.675	**11.31**	20.91	14.09	160.83	500_5d	**3.053**	159.97	**154.91**	42282.41	999801.74
200_5e	1231.049	201.39	207.33	**100.74**	277.90	500_5e	**2526.657**	**0.39**	6.41	7.77	176.26
200_10c	**12785.776**	**2.74**	23.75	34.85	150.40	500_10c	14567.870	19.45	21.82	**16.04**	60.34
200_10d	**17390.299**	**3.05**	7.60	11.92	43.43	500_10d	12875.890	**0.12**	2.69	13.25	86.87
200_10e	**17794.136**	**0.03**	0.05	2.60	63.35	500_10e	**15527.300**	**4.07**	11.73	9.50	64.39
200_15c	**30467.014**	**0.05**	0.30	7.79	100.80	500_15c	27109.630	**0.08**	6.49	23.23	55.07
200_15d	**22837.984**	**16.25**	42.28	47.55	118.27	500_15d	29550.430	**0.31**	7.97	10.06	85.27
200_15e	**28602.541**	**3.10**	17.88	19.96	100.96	500_15e	25010.130	**20.61**	25.06	30.11	106.69
200_20c	**41990.638**	**9.37**	19.81	25.52	83.76	500_20c	**37587.384**	**10.89**	25.91	32.81	87.31
200_20d	**41177.738**	**4.52**	20.56	26.18	83.11	500_20d	**43249.879**	**5.36**	10.22	15.77	49.51
200_20e	**39411.579**	**11.22**	21.00	34.81	84.52	500_20e	**39011.410**	**11.10**	21.95	22.43	96.33
Average Rank		**1.24**	2.24	2.52	4.00	**Average Rank**		**1.14**	2.24	2.76	3.86
Stat. Comp. vs MADEB			+	+	+	**Stat. Comp. vs MADEB**			+	+	+



(a) (b)

Fig. 1: Average ranks aggregated by the values of (a) n and (b) d

The results provided in Table 2 clearly show that MADEB outperforms all the other algorithms. In particular, our algorithm reached the best objective value in 76 instances over 126. Moreover, MADEB obtained the best ARDP value on 106 instances, i.e., about the 84% of the benchmark suite. Importantly, the statistical test has detected noticeable differences between our algorithm and the competitors: in every group of instances, MADEB is significantly better than all the other algorithms. Indeed, the worst Finner p-value (obtained in the comparison with GRASP+ePR on the group of instance $n = 200$) is about $2 \cdot 10^{-6}$, which is well below the commonly considered critical threshold 0.05 [24].

Finally, the better performances of MADEP with respect to its competitors are apparent also from the data provided in the two graphs of Figures 1a and 1b. Clearly, MADEB always obtained the best average rank, both when the instances are grouped by the values of n or the values of d.

7 Conclusions and Future Work

In this paper we have described a Memetic Algebraic Differential Evolution algorithm for solving Binary combinatorial optimization problems (MADEB). The main components of MADEB are: a binary algebraic differential mutation, and a variable neighborhood descent procedure. In particular, the binary differential mutation has been introduced as an instantiation of the abstract algebraic framework [1, 26–28].

MADEB has been applied to the MDTWNP problem. Experiments have been held on widely adopted benchmark instances, and the experimental results show that our approach has outperformed the state-of-the-art MDTWNPP algorithms.

As a future lines of research we are considering to apply MADEB to other binary optimization problems, like the knapsack problem and its variations, and to develop an algebraic particle swarm scheme [29] for the binary space.

Another possible future work is to extend MADEB for solving the multi-way generalization of MDTWNPP, as defined in [6], which would however require a major change to MADEB, because the solutions are no more binary strings.

Acknowledgement

The research described in this work has been partially supported by: the research grant "Fondi per i progetti di ricerca scientifica di Ateneo 2019" of the University for Foreigners of Perugia under the project "Algoritmi evolutivi per problemi di ottimizzazione e modelli di apprendimento automatico con applicazioni al Natural Language Processing"; and by RCB-2015 Project "Algoritmi Randomizzati per l'Ottimizzazione e la Navigazione di Reti Semantiche" and RCB-2015 Project "Algoritmi evolutivi per problemi di ottimizzazione combinatorica" of Department of Mathematics and Computer Science of University of Perugia.

References

1. Santucci, V., Baioletti, M., Milani, A.: Algebraic differential evolution algorithm for the permutation flowshop scheduling problem with total flowtime criterion. IEEE Transactions on Evolutionary Computation 20(5), 682–694 (2016), doi:10.1109/TEVC.2015.2507785
2. Kojić, J.: Integer linear programming model for multidimensional two-way number partitioning problem. Computers & Mathematics with Applications 60(8), 2302 – 2308 (2010)
3. Mertens, S.: The easiest hard problem: Number partitioning. Computational Complexity and Statistical Physics 125(2), 125–139 (2006)
4. Corus, D., Oliveto, P.S., Yazdani, D.: Artificial immune systems can find arbitrarily good approximations for the np-hard partition problem. In: Proc. of 15th Int. Conf. on Parallel Problem Solving from Nature - PPSN XV - Part II. pp. 16–28 (2018)
5. Rodriguez, F.J., Glover, F., García-Martínez, C., Martí, R., Lozano, M.: Grasp with exterior path-relinking and restricted local search for the multidimensional two-way number partitioning problem. Computers & Operations Research 78, 243 – 254 (2017)
6. Pop, P.C., Matei, O.: A memetic algorithm approach for solving the multidimensional multi-way number partitioning problem. Applied Mathematical Modelling 37(22), 9191 – 9202 (2013)
7. Kratica, J., Kojić, J., Savić, A.: Two metaheuristic approaches for solving multidimensional two-way number partitioning problem. Computers & Operations Research 46, 59 – 68 (2014)
8. Santucci, V., Baioletti, M., Milani, A.: A differential evolution algorithm for the permutation flowshop scheduling problem with total flow time criterion. In: Proc. of 13th International Conference on Parallel Problem Solving from Nature – PPSN XIII. pp. 161–170. Springer, Cham (2014), doi:10.1007/978-3-319-10762-2_16
9. Santucci, V., Baioletti, M., Milani, A.: Solving permutation flowshop scheduling problems with a discrete differential evolution algorithm. AI Communications 29(2), 269–286 (2016), doi:10.3233/AIC-150695

10. Santucci, V., Baioletti, M., Milani, A.: An algebraic differential evolution for the linear ordering problem. In: Companion Material Proceedings of Genetic and Evolutionary Computation Conference, GECCO 2015. pp. 1479–1480 (2015), doi:10.1145/2739482.2764693
11. Baioletti, M., Milani, A., Santucci, V.: Linear ordering optimization with a combinatorial differential evolution. In: Proc. of 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015. pp. 2135–2140 (2015), doi:10.1109/SMC.2015.373
12. Baioletti, M., Milani, A., Santucci, V.: An extension of algebraic differential evolution for the linear ordering problem with cumulative costs. In: Proc. of 14th International Conference on Parallel Problem Solving from Nature - PPSN XIV. pp. 123–133 (2016), doi:10.1007/978-3-319-45823-6_12
13. Baioletti, M., Milani, A., Santucci, V.: MOEA/DEP: An algebraic decomposition-based evolutionary algorithm for the multiobjective permutation flowshop scheduling problem. In: Proc. of European Conference on Evolutionary Computation in Combinatorial Optimization - EvoCOP 2018. pp. 132–145. Springer International Publishing, Cham (2018), doi:10.1007/978-3-319-77449-7_9
14. Baioletti, M., Milani, A., Santucci, V.: Learning bayesian networks with algebraic differential evolution. In: Proc. of 15th Int. Conf. on Parallel Problem Solving from Nature – PPSN XV. pp. 436–448. Springer International Publishing, Cham (2018), doi:10.1007/978-3-319-99259-4_35
15. Wang, L., Fu, X., Mao, Y., Menhas, M.I., Fei, M.: A novel modified binary differential evolution algorithm and its applications. Neurocomputing 98, 55 – 75 (2012)
16. Pampara, G., Engelbrecht, A.P., Franken, N.: Binary differential evolution. In: 2006 IEEE International Conference on Evolutionary Computation. pp. 1873–1879 (2006)
17. Storn, R., Price, K.: Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization 11(4), 341–359 (Dec 1997)
18. Milani, A., Santucci, V.: Asynchronous differential evolution. In: 2010 IEEE Congress on Evolutionary Computation (CEC 2010). pp. 1–7 (2010), doi:10.1109/CEC.2010.5586107
19. Price, K., Storn, R.M., Lampinen, J.A.: Differential evolution: a practical approach to global optimization. Springer Science & Business Media (2006)
20. Das, S., Suganthan, P.N.: Differential evolution: A survey of the state-of-the-art. IEEE Transactions on Evolutionary Computation 15(1), 4–31 (Feb 2011)
21. Das, S., Mullick, S.S., Suganthan, P.: Recent advances in differential evolution – an updated survey. Swarm and Evolutionary Computation 27, 1 – 30 (2016)
22. Brest, J., Greiner, S., Boskovic, B., Mernik, M., Zumer, V.: Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. IEEE Transactions on Evolutionary Computation 10(6), 646–657 (Dec 2006)
23. Pavai, G., Geetha, T.V.: A survey on crossover operators. ACM Computing Surveys 49(4), 72:1–72:43 (Dec 2016)
24. Derrac, J., García, S., Molina, D., Herrera, F.: A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm and Evolutionary Computation 1(1), 3 – 18 (2011)

25. Ceberio, J., Irurozki, E., Mendiburu, A., Lozano, J.A.: A distance-based ranking model estimation of distribution algorithm for the flowshop scheduling problem. IEEE Transactions on Evolutionary Computation 18(2), 286–300 (April 2014)
26. Baioletti, M., Milani, A., Santucci, V.: Algebraic particle swarm optimization for the permutations search space. In: Proc. of 2017 IEEE Congress on Evolutionary Computation (CEC 2017). pp. 1587–1594 (2017), doi:10.1109/CEC.2017.7969492
27. Baioletti, M., Milani, A., Santucci, V.: Automatic algebraic evolutionary algorithms. In: Proc. of Int. Workshop on Artificial Life and Evolutionary Computation (WIVACE 2017). pp. 271–283. Springer International Publishing, Cham (2018), doi:10.1007/978-3-319-78658-2_20
28. Baioletti, M., Milani, A., Santucci, V.: Algebraic crossover operators for permutations. In: 2018 IEEE Congress on Evolutionary Computation (CEC 2018). pp. 1–8 (2018), doi:10.1109/CEC.2018.8477867
29. Santucci, V., Milani, A.: Particle swarm optimization in the EDAs framework. In: Soft Computing in Industrial Applications. pp. 87–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2011), doi:10.1007/978-3-642-20505-7_7