A Memetic Algorithm for the Weighted Feedback Vertex Set Problem

Francesco Carrabs*, Carmine Cerrone and Raffaele Cerulli

Department of Mathematics, University of Salerno, Italy.
fcarrabs@unisa.it, ccerrone@unisa.it, raffaele@unisa.it

Abstract

Given an undirected and vertex weighted graph $G = (V, E, w)$, the Weighted Feedback Vertex Set Problem consists of finding the subset $F \subseteq V$ of vertices, with minimum weight, whose removal results in an acyclic graph. Finding the minimum feedback vertex set in a graph is an important combinatorial problem that has a variety of real applications. In this paper we introduce a memetic algorithm for this problem. We propose an efficient greedy procedure that quickly generates chromosomes with specific characteristics and a wise application of a recent local search procedures based on k-diamonds. Computational results show that the proposed algorithm outperforms the effectiveness of two other metaheuristics recently proposed in the literature for this problem.

Keywords: Memetic Algorithm, Feedback Vertex Set, Loop Cutset, k-diamond.

1. Introduction

Given an undirected graph $G = (V, E)$, a *feedback vertex set* (fvs) $F \subseteq V$ of G is a subset of vertices whose removal results in an acyclic graph (a forest). The feedback vertex set problem (FVS) consists of finding a fvs of G with minimum cardinality. If G is a vertex weighted graph, then we have the weighted version of problem (WFVS) that consists of finding the fvs of G with minimum weight.

This problem finds applications in many fields. For example, in the context of operating systems, it models the problem to prevent and/or remove deadlocks [26] generated by cyclical processes requests of already locked resources. A similar problem arises in the context of combinatorial circuit design where the circuits are represented by graphs in which a cycle could generate a "race condition", that is, some

*Corresponding author.

circuit elements could receive new inputs before being stabilized. Another application concerns the study of "monopolies" in synchronous distributed systems [22, 23] where the connection networks are represented by grid and toroidal graphs. Other applications include program verification [24] and constraint satisfaction problems [2].

The feedback vertex set problem has been extensively studied (see [10] for a recent survey), and it is among the first problems shown to be NP-complete [14]. However, it results solvable in polynomial time on particular class of graphs like: k-diamond graphs [6, 7], permutation graphs [16, 25], reducible flow graphs [18], interval graphs [19], co-comparability graphs and convex bipartite graphs [13, 17].

For the cases that are not known to be polynomially solvable there have been intensive efforts on approximation algorithms [1, 2, 3, 9, 15] whereas very few heuristics are proposed in the literature for the WFVS. To the best of our knowledge, for the FVS problem a GRASP procedure [21] and a simulated annealing algorithm [12] are introduced whereas two metaheuristics XTS [4] and ITS [7] are proposed for the WFVS. The tabu search XTS is based on the "eXploring Tabu Search" schema [8]. Thanks to this schema, XTS finds solutions very close to the optimal one in few seconds. In this metaheuristic the neighborhoods are represented by k-diamond graphs generated by moving a single vertex from the fvs to the forest. These neighborhoods are explored using an approximation algorithm [1]. The computational complexity of finding an optimal fvs on a k-diamond graph was left as an open question. This open question is solved in [5, 6] where a linear time procedure is proposed. The authors embedded this procedure into an iterative tabu search ITS for the WFVS problem [7].

Looking at neighborhoods used by XTS and ITS algorithms, we can observe that they essentially perform vertex exchanges between the fvs and the residual graph induced by removing the fvs from the graph. Due to the WFVS problem characteristics, the generation of new neighborhoods, different from simple vertices exchange, does not appear to be a simple task. In this contest, a genetic approach, in which several fvs are created by mixing the vertices of the existing ones, appears to be particularly suitable. For this reason we propose a memetic algorithm whose main aim is to be more effective than the other metaheuristics for WFVS. The accomplishment of our aim is certified by computational results carried out on benchmark instances. These results show that our algorithm, on the small instances, often finds the optimal solution. Moreover, on the large instances, our algorithm finds better solutions than ITS and XTS algorithms.

The remainder of the paper is organized as follows. Section 2 introduces the definitions and notations that are used throughout the paper. Section 3.1.1 contains the description of our greedy algorithm for the creation of chromosomes. The memetic algorithm is described in Section 4. Finally, the computational results are presented in Section 5 and some concluding remarks are given in Section 6.

2. Definitions and Notations

Let $G = (V, E, w)$ be an undirected and vertex weighted graph, where V is the set of n vertices, E is the set of m edges, and, $w(v)$ is a positive weight associated with each vertex $v \in V$. Given a subset $X \subseteq V$ of vertices, we define $\bar{X} = V \setminus X$ and $W(X)$ as the sum of the weights of its elements, i.e. $W(X) = \sum_{v \in X} w(v)$. We denote by $G[X]$ the subgraph of G induced by the set of vertices $X \subseteq V$. Formally, $G[X] = (X, E_{[X]}, w)$ where $E_{[X]} = \{(u, v) \in E : u, v \in X\}$. Let $\delta_X(v)$ and $d_X(v)$ be the set of vertices adjacent to v and the degree of the vertex v in $G[X]$, respectively. When $X = V$, we simply denote these data by $\delta(v)$ and $d(v)$, respectively. Moreover, let $rnd(a, b)$ be a function that returns a random integer value within the interval $[a, b]$.

A *tree* is an acyclic and connected graph while a *forest* is an acyclic graph in which any connected component is a tree. Given a set of vertices $X \subseteq V$, the *residual graph* of G, generated by X, is the subgraph $G[\bar{X}]$. The set X is a feedback vertex set (fvs) of G if the residual graph $G[\bar{X}]$ is a forest. For the graph G depicted in Figure 1a, the set of vertices $X = \{6, 16\}$ is a fvs of G because $G[\bar{X}]$ is a forest (Figure 1b).

From now on, we denote by $F(G)$ and by $F^*(G)$ any fvs and a minimum weight fvs (*optimal solution*) of G, respectively. When no confusion may arise, we simply denote these sets by F and F^*, respectively. A vertex $v \in F(G)$ is *redundant* if $F(G) \setminus \{v\}$ is again a fvs of G. $F(G)$ is a *minimal* fvs if it does not contain redundant vertices, i.e. there can not be a fvs $F'(G)$ such that $|F'(G)| < |F(G)|$ and $F'(G) \subset F(G)$. Any vertex $v \in V$ is said to be *appended* if it is not included in any cycle of G. In Figure 1c the vertices 2, 9 and 14 are appended. We say that a graph is *reduced* if it does not contain any appended vertex (Figure 1d). Finally let us define the k-diamond graphs [7].

Definition 1. A weighted *k-diamond* $\mathcal{D}_{R,z} = (V_R, E_R, w)$, where $k \geq 1$, $R = \{r_1, r_2, \ldots, r_k\}$ and $z \in V_R$, is an undirected and vertex weighted graph where *(i)* each vertex $v \in V_R$ is included in a simple path between exactly one of the k apices $r_i \in R$ and the vertex z and *(ii)* $\mathcal{D}_{R,z}[V_R \setminus \{z\}]$ is a forest with k connected components.

In Figure 1d the 2-diamond graph $\mathcal{D}_{\{5,7\},6}$ is depicted, where $r_1 = 5$, $r_2 = 7$ and $z = 6$. As we will see in section 3.5, our local search procedure builds k-diamonds graphs as subgraphs of G and it uses them to improve the solutions. For instance, the k-diamond graph $\mathcal{D}_{\{5,7\},6}$ shown in Figure 1d can be obtained from G by removing the vertex 16 and the appended vertices generated by its removal.

To simplify the notation, in the rest of the paper we denote a diamond $\mathcal{D}_{R,z}$ just as $\mathcal{D}_z$, since for our aims the upper apices are negligible.

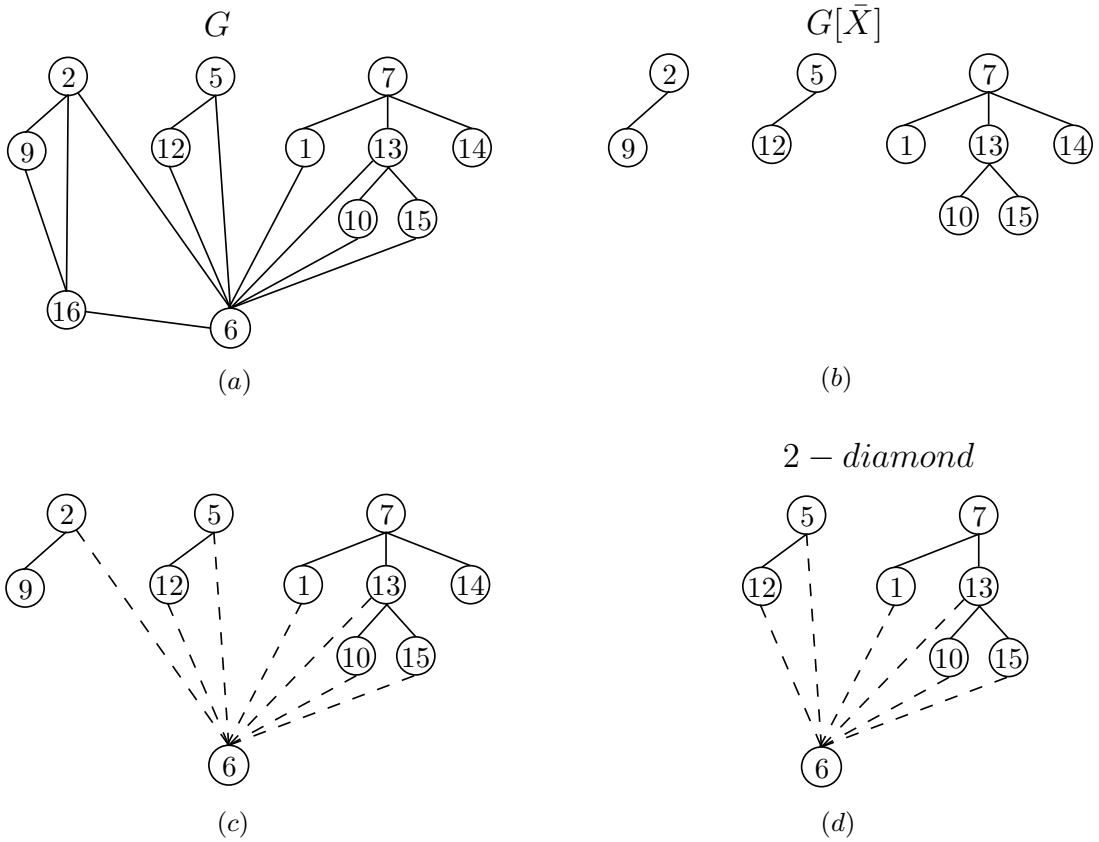

Figure 1: *(a) A generic graph G. (b) The residual graph $G[\bar{X}]$ where $X = \{6, 16\}$. Since $G[\bar{X}]$ is a forest, X is a fvs of G. (c) The subgraph of G obtained reinserting the vertex 6 in the residual graph $G[\bar{X}]$. This graph contains three appended vertices: 2,9,14. (d) Removing these vertices we obtain a 2-diamonds graph.*

3. A memetic approach: basic components

Memetic algorithms belong to the class of evolutionary algorithms that use local search within a classical genetic algorithm framework to intensify the search phase. The algorithm starts with a set of solutions for the WFVS problem (represented by chromosomes), called population, and it uses the solutions in this population to generate a new population. In particular, at each iteration, two chromosomes (*parents*) are selected from the population and the crossover operator is invoked on these parents to generate a new chromosome (*child*) on which the mutation operator is applied. Finally, a local search, based on k-diamonds, is applied on the child before it replaces one of the two parents into the population. This is motivated by the hope that the new population will be better than the previous one. Solutions which are selected to form new solutions are selected mainly according to their fitness: the more suitable they are the more chances they have to be

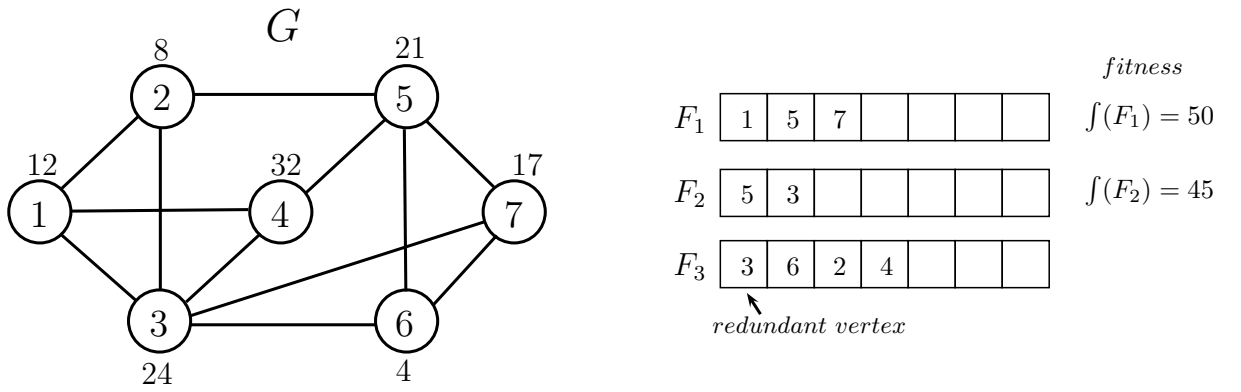

Figure 2: *Three fvs of graph G. F_1 and F_2 are chromosomes because they are minimal fvs of G. Their fitness is obtained by adding the weight of their vertices (there are no penalties in this example). F_3 is not a chromosome because the vertex 3 is redundant.*

selected for the crossover operation. This is repeated until a stop criterion is fulfilled. For a complete and detailed description of the memetic algorithms and their characteristics the reader can refer to [20].

The elements that compose our memetic algorithm are described below in details.

3.1 Chromosomes and fitness

We define the chromosome as a set $F \subseteq V$ of vertices (each vertex in F can be viewed as a gene) such that $G[\bar{F}]$ is a forest and there are no redundant vertices in F. In other words, a chromosome is a minimal fvs of the graph. The fitness $f(F)$ of chromosome F is defined as the sum of weight $w(v)$ and (possible) penalty $\rho(v)$ of each vertex v in F. Formally: $f(F) = \sum_{v \in F}\{w(v) + \rho(v)\}$.

In Figure 2 a graph G and three fvs of graph are shown. F_1 and F_2 are chromosomes of G and their fitness value is shown on the right. In this example, the fitness value is equal to the sum of the weights of the vertices because there are no penalties associated to the vertices. The last fvs F_3 is not a chromosome because, by removing the vertex 3 from F_3, we obtain a smaller fvs of G. As we will see later, the sets of vertices generated by crossover and mutation operators can contain redundant vertices. To avoid this, every time a new fvs is created or modified by these operators, a redundancy control is carried out by our algorithm.

3.1.1 Chromosomes generation: the Snd procedure

In this section we introduce the procedure we used to generate chromosomes, namely the Snd procedure. Given a graph $G = (V, E, w)$, Snd computes a fvs F of G according to the pseudocode shown in Algorithm 1. At the beginning the set of vertices X, of the residual graph, is equal to V while F is an empty set (line 1).

At each iteration, the algorithm randomly selects a vertex $u \in X$ among the three vertices with lowest ratio $\Delta_X(u) = w(u)/ND(u)$ where $ND(u) = \sum_{v \in \delta(u)} w(v)/\sqrt{d_X(v)}$. The vertex u is moved from residual graph $G[X]$ to F. Then the algorithm removes all the appended vertices in $G[X \setminus \{u\}]$. The process is repeated until $X = \emptyset$. Finally, any redundant vertex in F is removed.

Algorithm 1: Snd

Input: $G(V, E, w)$;
Output: a feedback vertex set of G;

```
1  X ← V, F ← ∅;
2  Δ_X(u) ← w(u) / Σ_{v∈δ(u)} w(v)/√d_X(v)        ∀v ∈ X;
3  while X ≠ ∅ do
4  |   Randomly select a vertex u among the three vertices with lowest Δ_X;
5  |   X ← X \ {u}, F ← F ∪ {u};
6  |   while ∃v ∈ X : d_X(v) < 2 do
7  |   |   X ← X \ {v};
8  |   Update d_X(u) and Δ_X(u) in G[X]        ∀u ∈ X;
9  Remove redundant vertices from F;
10 return F;
```

The Snd is a greedy procedure that, by using the value of Δ_X, at each iteration tries to identify the *more promising* vertex to be added to the fvs. The more promising vertex is the one having lowest ratio between its weight and the number of cycles that are broken by its removal from the graph. However, since the number of broken cycles cannot be computed in polynomial time, we use an estimate of this number. For each vertex $u \in X$, Snd computes $\Delta_X(u) = w(u)/ND(u)$ and classifies, as the more promising vertex, the one with the lowest $\Delta_X(u)$. Therefore, the probability to select the vertex u is inversely proportional to its weight and directly proportional to $ND(u) = \sum_{v \in \delta(u)} w(v)/\sqrt{d_X(v)}$. This last value represents an estimate of the neighborhood of u that takes into account the weight and degree of all vertices in $\delta(u)$. When the value of $ND(u)$ increases, the probability that the removal of u is more convenient that the removal of any other vertex $v \in \delta(u)$ increases.

The computation of $ND(u)$ is carried out using the squared root of its degree in order to reduce the impact of degree on the ratio $w(v)/\sqrt{d_X(v)}$; this is particularly important when the vertex weights have low values.

We will use the Snd procedure to generate the initial population, for our memetic algorithm, but also to transform (removing the redundancy) the set of vertices returned by the crossover operator into a chromosome (when needed).

3.2 Initial Population

The initial population is composed of σ chromosomes. Since the performance of our algorithm is affected by population size, we chosen $\sigma = 50$ because this is the minimum value that, experimentally, guarantees us the individuation of good quality solution without penalizing the performance. In order to ensure a better dispersal of solutions and to reduce the risk of a premature convergence, the algorithm tries to avoid the presence of *clones* (identical chromosomes) in the initial population. Every time a clone is generated, the algorithm invokes the mutation operator, described in section 3.4. The generation of chromosomes is carried out by randomly selecting one of the two methods described next.

The first method invokes the Snd procedure on the graph G. Each execution of Snd often produces a different chromosome of the graph because of the vertex selection policy applied by Snd at each iteration (Algorithm 1, line 4).

The second method builds a fvs by randomly selecting the vertices, at each iteration. In more detail, the second method starts with a residual graph $G' = G$ and $F \leftarrow \emptyset$ and, at each iteration, the procedure randomly selects a vertex $v \in V'$ that is removed from G' with its incident edges. The residual graph G' is updated consequently (i.e. $V' \leftarrow V' \setminus \{v\}$, $E' \leftarrow E' \setminus \{ \bigcup_{k \in \delta_{V'}(v)} (v,k)\}$) and the vertex v is inserted in F. Then the procedure recursively removes from G' all vertices whose degree has become less than 2, due to the removal of the vertex v, because they do not belong to cycles. The procedure stops when $V' = \emptyset$. The resulting set of vertices F is a fvs of G. Since the operations, carried out at each iteration, have a cost equal to the number of edges removed, the total cost of the procedure is $O(m)$.

Since the chromosomes, obtained by the second method, contain any kind of vertex regardless of its characteristics (weight, degree, etc.), the presence of these chromosomes into the population guarantees that all the vertices of the graph can participate to the evolutionary process.

The application of these two methods allows the creation of a variegated population that will benefit of a better exploration of the solutions space.

3.3 Crossover operator

The crossover operator plays a key role in the memetic algorithm because it influences the effectiveness of the algorithm. This operator allows the creation of new chromosomes by using the chromosomes present in the population. In particular, the crossover takes in input two chromosomes of the population, the parents, and generates a new chromosome, the child, that bears a resemblance to each parent.

In details, the chromosomes in the population are sorted, in ascending order, according to their fitness values. The first parent, F_b, is randomly selected from the whole population whereas the second parent, F_a, is randomly selected among the chromosomes with fitness lower than or equal to $f(F_b)$. The crossover uses F_a and F_b to generate a new set of vertices, F_c, by randomly selecting k different vertices from the two parents, where $k = min\{|F_a|, |F_b|\}$. If F_c is a fvs of G, then the operator performs a redundancy control

on it and returns the new chromosome. Otherwise a modified version of Snd procedure on the graph $G[\bar{F}_c]$ is invoked. This modified version builds a new chromosome by starting from a set of vertices F_c instead of starting from an empty set. To this end, it is sufficient to change the line 1 in the Algorithm 1 as follows: $X \leftarrow V \setminus F_c$ and $F \leftarrow F_c$.

The aim of our crossover is to generate children by joining the vertices of the parents without taking in account neither the characteristics of these vertices like the weight, the degree and the occurrences into the population nor the fitness of children produced. The vertices are selected randomly from the parents and, as a result, each vertex has the same probability to be inherited from the children. In this way, we try to keep a heterogeneous population in order to avoid the problem of a fast convergence of the algorithm towards a local minimum. In this step, we do not care about the fitness of children produced by the crossover, because the goal to improve this fitness is delegated to the local search procedure.

3.4 Mutation Operator

Mutation is a genetic operator that alters one or more genes in a chromosome to introduce perturbation and thus providing diversification in the new generated chromosomes. Our mutation operator modifies a chromosome F by replacing some of its vertices with vertices of the forest $G[\bar{F}]$. More in detail, the mutation randomly selects and removes k vertices from F, where $k = rnd(1, 5)$. The resulting set of vertices $F' \subset F$ is not a fvs of G because, by definition, F does not contain redundant vertices. The k removed vertices are inserted into the forest $G[\bar{F}]$ and a new graph $G' = (V', E_{[V']}, w)$, where $V' = V \setminus F'$, that contains cycles is built. For each cycle in G', one of its vertices is removed by the mutation operator until an acyclic graph is obtained. The vertex to remove from cycle is selected by applying, randomly, one of the following three rules:

- Select the vertex v with minimum ratio $w(v)/d(v)$;
- Select the vertex v with maximum degree $d(v)$;
- Randomly select a vertex v of the cycle;

The selected vertex v is then added to F'. Note that, according to the previous three rules, the selected vertex v could be one of the k vertices previously removed from F. Since the aim of the mutation operator is to generate a solution different from F, the operator selects one of these k vertices only if it is strictly necessary, i.e. if there is a cycle in G' composed only of vertices in $F \setminus F'$. The selection process is repeated until no more cycles are into the residual graph G'. Finally, a redundancy control on F' is carried out.

3.5 Local Search procedure

To enforce our genetic algorithm we use a local search procedure called *best local search* (BLS). The idea behind the BLS is to reduce the weight of the chromosome by "replacing" one of its vertices with a cheaper

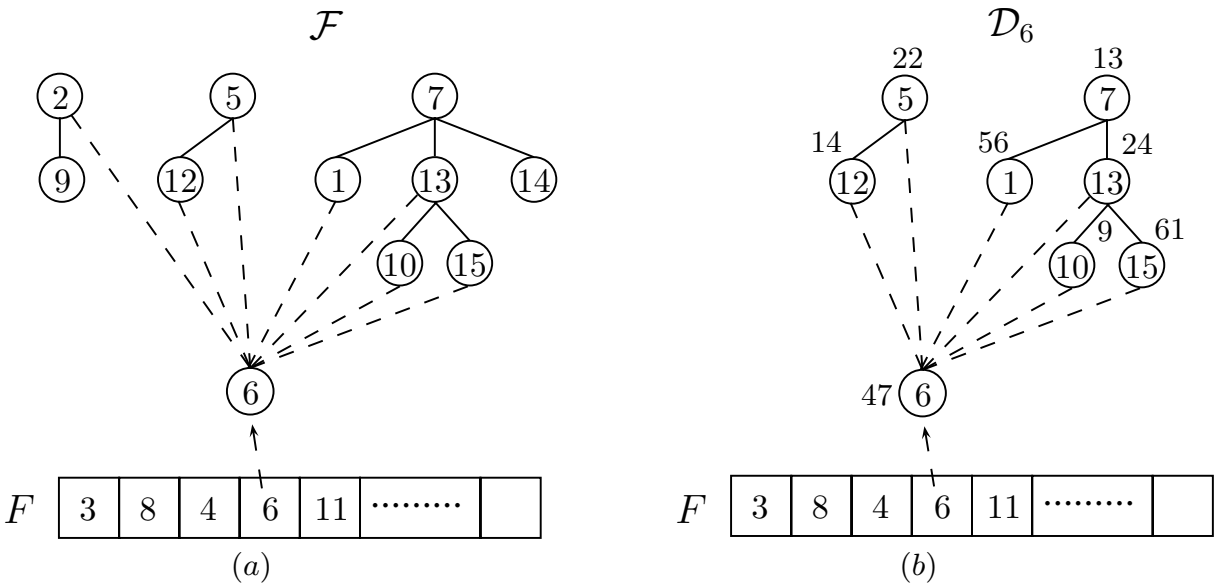

Figure 3: *a) The vertex 6 of the chromosome is introduced into the forest $\mathcal{F}$ generating cycles. b) By removing all the appended vertices from $\mathcal{F} \cup \{v\}$ a 2-diamond graph $\mathcal{D}_6$ is obtained. The optimal fvs of $\mathcal{D}_6$ is $F_6 = \{7, 10, 12\}$ with $W(F_6) = 36$. Since $w(v) = 47 > W(F_6)$, by replacing the vertex 6 with F_6 in F a new chromosome with lower weight is obtained.*

set of vertices from the forest. In more details, let F be a chromosome of G and let $\mathcal{F} = G[\bar{F}]$ be the forest induced by vertices outside the chromosome. It is easy to see that moving a vertex v from F to $\mathcal{F}$, we generate a new graph $G' = \mathcal{F} \cup \{v\}$ in which all the cycles pass through v (see Figure 3a). By removing from G' all the appended vertices a 2-diamond graph $\mathcal{D}_v$ is obtained (Figure 3b). The WFVS problem can be solved in linear time on the k-diamond graphs by using the dynamic programming algorithm (DP) provided in [7].

Let us see now how the BLS uses the DP algorithm to decrease the weight of chromosomes, when possible. For each vertex $v \in F$, the local search procedure inserts v into the forest $\mathcal{F}$ and it removes all the appended vertices from $\mathcal{F} \cup \{v\}$ obtaining the k-diamond $\mathcal{D}_v$. Then the BLS invokes the DP algorithm on $\mathcal{D}_v$ to find the optimal fvs F_v. If $W(F_v) \geq w(v)$ (i.e. no improvements are possible by replacing v) the BLS rebuilds the forest $G[\bar{F}]$ and selects the next vertex of chromosome. Otherwise the BLS replaces v with F_v in F, produces a new chromosome F', with $W(F') < W(F)$, and invokes a redundancy control on F'. The BLS compares the improvement $W(F) - W(F')$, obtained by replacing v, with the best improvement found so far and it saves the replacement with greatest improvement between them. Finally, the BLS rebuilds the forest $G[\bar{F}]$ and selects the next vertex. The local search repeats this process for all the vertices of F and, at the end, it carries out the best replacement on the chromosome F. As a result, a better chromosome F_1 is produced on which the BLS invokes itself recursively. The local search stops when it does not improve the

chromosome on which it is invoked.

The only drawback of the BLS is computational expensive. Indeed, this procedure invokes the DP algorithm for each vertex v belonging to the chromosome and, every time the replacement of v generates an improvement, a computational expensive redundancy control on the new chromosome F' must be carried out. Finally, every time the BLS generates a better chromosome, through the best replacement, the local search recursively invokes itself on this chromosome. For this reason, the BLS procedure has to be used in an appropriate manner not to slow down the performance of memetic algorithm.

For instance, the BLS should not be invoked on the *bad* chromosomes, i.e. the chromosomes with a weight that is at least 20% greater than the weight of the current best chromosome into the population. On this chromosome, the BLS usually carries out a lot of improvements that means a lot of recursive invocations. We solve this problem by introducing another local search procedure, named *first local search* (FLS), that carries out the first replacement that improves the chromosome instead of the best one as BLS does. This makes the FLS much faster than the BLS and more useful to improve the bad chromosomes. The BLS will be used on the *good* chromosomes, i.e. the chromosomes with a weight that is at most 20% greater than the current best chromosome. Usually, on these chromosomes there are few possible improvements and so we prefer to apply the BLS to gain the maximum improvement, every time a replacement is carried out by procedure during the computation.

4. The Memetic Algorithm

In this section we introduce our memetic algorithm (MA) based on all the elements described in previous sections. The MA combines the exploration capabilities of genetic algorithms with efficient local search procedures and with a diversification mechanism based on a penalization of the vertices. In Algorithm 2 the pseudocode of MA is shown.

The first step carried out by MA is the generation of an initial population and the identification of its best chromosome $\hat{F}$ (line 2). The *repeat-until* loop (lines 3-25) manages the diversification mechanism that is applied max_div times. Since the algorithm could be trapped into a local minimum, far from the optimal solution, we overcome this event by applying the diversification schema that generates a new initial population taking in account the penalizations associated with the vertices of the local minimum. In this way, MA can improve the current incumbent solution by exploring new regions in the solutions space. Obviously, as the value of max_div increases, the chances to find better solutions increase. In our implementation max_div is set to 1 because the improvements gained in our test results, with greater values of max_div, do not justify the relevant increment of computational time of algorithm.

The body of the *repeat-until* loop is divided into two phases. The first phase (lines 4-21) represents the core of memetic algorithm in which the classical operations of such approach are carried out. The second phase (lines 22-24) is used to diversify the chromosomes of the population by penalizing the vertices in $\hat{F}$.

Algorithm 2: MA

Input: $G(V, E, w)$;
Output: a minimal feedback vertex set of G;

```
1  n_div ← 0; current_iter ← 0;
2  Build the initial population P and let F̂ be the best chromosome in it;
3  repeat
4      while current_iter ≤ MaxIt do
5          for i ← 1 to (0.2 × |P|) do
6              Select parent chromosomes F_a and F_b;
               // w.l.o.g. let F_b be the worst parent, i.e. f(F_a) < f(F_b) and
               //          F_b be different from parents chosen in the previous steps of the for loop
7              F_c ← Crossover(F_a, F_b);
8              if Chk_Fitness(F_c) = false then
9                  F_m ← Mutation(F_c);
10                 if Chk_Fitness(F_m) = true then
11                     F_c ← F_m;
12                 else
13                     if f(F_m) < f(F_c) then
14                         replace the worst parent F_b with F_m in P;
15                     else
16                         replace the worst parent F_b with F_m or F_c in P with the same probability;
17             if current_iter ≥ 0.8 × MaxIt then
18                 if (f(F_c) ≤ f(F̂) + 0.2 × f(F̂)) then
19                     F_c ← BLS(F_c);
20                 else
21                     F_c ← FLS(F_c);
22     Penalize(F̂);        // Penalize the 75% of vertices in F̂
23     P ← New_Pop(P);      // Modify the current population according to the penalizations
24     UnPenalize(F̂);       // Remove penalizations
25 until n_div > max_div;
26 return F̂;

27 Function: Chk_Fitness(F_c)
28 if f(F_c) < f(F̂) then
29     F̂ ← F_c; current_iter ← 0, n_div ← 0;
30     Replace the worst parent F_b with F_c in P;
31     return true;
32 else
33     return False
```

In the sections to follow a detailed description of these two phases is given.

4.1 The first phase

The first phase of MA consists of a *while* loop (lines 4-21) repeated for $MaxIt$ consecutive iterations, without improvement of the incumbent solution $\hat{F}$. The value of $MaxIt$ is computed by means of the

following formula: $MaxIt = 50 + 200/(\sqrt{n} \times \sqrt{D(G)})$ where $D(G)$ is the density of the graph. The value 50 in the formula represents the minimum number of iterations that MA has to execute. The value of the remaining part of the formula depends on the size and on the density of G. In particular, the value of $MaxIt$ decreases, as the size and the density of the graph increases. We made this choice to improve the performance of MA that depends on these two characteristics of the graph. On the small instances (up to 75 vertices), the formula assigns a high value to $MaxIt$ to increase the chances to find the optimal solution. Since the iterations are very fast on this kind of instances, no performance problems arise. Instead, on large instances the iterations are more expensive and therefore the algorithm has to perform less iterations to avoid poor performance. Although this choice reduces the effectiveness of our algorithm on large instances, the results of computational tests show that MA remains more effective than the other metaheuristics.

At each iteration of the first phase (line 4) we want to generate a new population that differs from the previous one by at most 20% of chromosomes. To this end, we use a for loop (line 5) executed $0.2 \times |P|$ times. In this loop the selection of parents, the crossover operator, the mutation operator and the local search procedures are sequentially executed and a new child chromosome is inserted into the population, when possible. The first step of the for loop is the selection of the two parent chromosomes, F_a and F_b, from the current population, on whose the crossover operator is applied to generate the child chromosome F_c (lines 6-7). W.l.o.g. we suppose that F_b is the worst of the two parents, that is $f(F_a) < f(F_b)$, that is different from any other chromosome chosen in the other steps of the for loop. On line 8, the $Chk\_Fitness$ procedure (lines 27-33) is invoked on F_c. This procedure checks if the fitness of F_c is lower than the fitness of the incumbent solution $\hat{F}$. If this is the case, the incumbent solution $\hat{F}$ is updated to F_c, the iteration and diversification counters are set to 0 and the parent F_b is replaced by child F_c (lines 29-31). Otherwise the procedure returns the false value in order to signal that F_c is not inserted into the population. Going back to the main loop, if $Chk\_Fitness$ returns false (line 8) then the mutation operator on F_c is applied (line 9). On the "mutated" child F_m the $Chk\_Fitness$ procedure is again invoked (line 10) and if returns true then F_m is assigned to F_c. Otherwise if $Chk\_Fitness$ returns false, the algorithm compares the fitness of F_m and F_c (line 13) to establish who is the chromosome that have to replace the worst parent F_b into the population. If $f(F_m) \leq f(F_c)$ then F_m replaces F_b otherwise one between F_m and F_c is randomly selected by the algorithm in order to replace F_b. This last choice is made because we prefer to apply the local search procedure on the mutated chromosome instead of the one generated by the crossover operator.

In lines 17-21 the criterion used to invoke the BLS and FLS are reported. The application of these two local search procedures represents the key of MA effectiveness. However, since this application is expensive in terms of computational time, it is necessary to decide accurately "how" and "when" to invoke these procedures, during the computation, to obtain a good trade-off between effectiveness and performance. To this end, the 80% of iterations are carried out by MA without the application of the local search procedures (line 17). We made this choice because we want to preserve the variety of the initial population. Under the threshold of 80% the evolution of population is carried out partially neglecting the fitness of the new

chromosomes generated. Only the invocation of Snd procedure, inside the crossover operator, takes in account the fitness value to "complete" the chromosome. In this way any kind of chromosome can be present into the population assuring us a high level of diversification. Instead, by applying the local search procedures from the beginning, we risk to lose this diversification level and to produce a premature convergence of the algorithm. The threshold of 80% is introduced also for performance reasons. As already explained in section 3.5, the application of local search procedures on bad chromosomes is expensive because of the high number of improvements carried out. For this reason it is not convenient to apply these procedures during the first iterations of the algorithm when there could be a lot of bad chromosomes in the population.

When the test of line 17 holds, if F_c is a good chromosome then BLS is invoked (line 19) otherwise FLS is invoked (line 21). The *if* on line 18 is used to classify the chromosomes in good and bad according to the definition given in section 3.5.

As we will see in section 5, the introduction of FLS procedure and the criterion used by MA to invoke the two local search, makes our algorithm much more effective than the ITS metaheuristic [7] that uses only the BLS local search.

4.2 The second phase: population replacement

Since, during the first phase, MA may get trapped into a local minimum, it is necessary to provide a mechanism to reduce this occurrence. To this aim, we use the chromosomes present, at the end of the first phase, in the population. Indeed, since many of these chromosomes have a fitness often close to $f(\hat{F})$, possible better solutions can be obtained by replacing few vertices. Given that MA has to escape from the current local minimum $\hat{F}$, the identification of the vertices to be replaced is carried out by adding a penalty to the vertices in $\hat{F}$. Then the first step of the second phase invokes the $Penalize$ procedure on $\hat{F}$ (Algorithm 2, line 22). This procedure randomly selects the 75% of vertices of $\hat{F}$ and it sets their penalties equal to $\max_{v \in V}\{w(v)\}$ while the remaining vertices of the graph have the penalties set to zero. The fitness of all the chromosomes in the population is recomputed according to the new penalties. Since the fitness of the chromosomes is given by $f(F) = \sum_{v \in F}\{w(v) + \rho(v)\}$, if a chromosome contains at least one of the vertices selected by the $Penalize$ procedure then its fitness value grows by penalizing it. The New_Pop procedure is invoked (line 23) on the population. This procedure implements the first phase (lines 4-21) with the only difference that the number of iterations carried out is very low ($MaxIt = 10$). Since the fitness of chromosomes is computed by adding the weight and the penalties of each vertex into the chromosome, the algorithm tends to avoid the penalized vertices for the creation of new chromosomes. As a result, the procedure generates a new population that, by using few vertices of $\hat{F}$, allows the exploration of a new part of the space solutions. At the end of New_Pop procedure, all the penalties are set to zero (line 24) and the counter of diversification is increased by one. The algorithm restarts from the new population just generated.

5. Test Results

The MA algorithm was coded in C++ and runs on a 2.33 GHz Intel Core2 processor. The tests are carried out on the set of benchmark instances proposed in [7]. Besides the generic random graphs, this benchmark set includes grid, toroidal and hypercube graphs whose practical applications are presented in [11, 22, 23]. Each instance is characterized by the number of vertices (size), the number of edges and a range of values for the weight of the vertices. The instances are divided into two groups: the small instances, composed of 25, 50 and 75 vertices, and the large instances, where the instance sizes range from 100 to 500 with increments of 100. The weight of vertices is an integer value selected within the ranges: 10-25, 10-50 and 10-75. The combination of all these parameters allows us to verify the robustness of the solution produced by the algorithms. In particular, we want to highlight how the effectiveness and performance of the algorithms are affected by problem size, by density of the graph, by the weight ranges and by the classes of graphs. We compare our MA algorithm with the two tabu search ITS [7] and XTS [4].

The XTS algorithm is based on the "eXploring Tabu Search" schema [8]. The basic idea behind this schema is to select the best solution into the neighborhood but in the same time to create a set of "good" solution of the neighborhood that could be used in the following as starting point to diversify the search, leading it toward promising regions. When some conditions are satisfied, the search restarts from one of these solutions. The effectiveness of this schema is proven by results obtained by XTS that, in few seconds, finds solutions very close to the optimal one. In this metaheuristic the neighborhoods are represented by k-diamond graphs generated by moving a single vertex from the fvs to the forest. These neighborhoods are explored using an approximation algorithm [1]. The ITS algorithm is an iterative tabu search that uses the same neighborhood of XTS but this last is explored in a more efficient way by replacing the approximation algorithm with a linear procedure that is able to solve the WFVS problem on the k-diamonds and then to always individuate the best solution inside the neighborhood. Since the authors of XTS and ITS provided us their source codes, an accurate comparison of these three algorithms is carried out because they are compiled and executed on the same machine.

5.1 Small Graphs

In Table 1 the results of the three algorithms on the small random graphs are reported. The first five columns report the id (id) of the instance, the number of vertices (n), the number of edges (m), the lower (low) and the upper (up) bounds of the vertex weight, respectively. The sixth column (Opt) shows the optimal solution value. The columns ITS, XTS and MA are divided into two subcolumns (Value and Time) reporting their solution value (in bold when equal to the optimal value) and the computational time in seconds, respectively. The results reported in each line are the average values over five instances with the same characteristics but with a different node weight assignment. The second last line reports, for each algorithm, its average gap value (agv) and its average computational time (avg_t). The average gap value is computed using the Opt

Random graphs: Small Instances											
Id	**Instance**				**Opt**	**ITS**		**XTS**		**MA**	
	n	**m**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
S_R1	25	33	10	25	63.8	**63.8**	0.00	**63.8**	0.00	**63.8**	0.08
S_R2	25	33	10	50	99.8	**99.8**	0.00	**99.8**	0.00	**99.8**	0.08
S_R3	25	33	10	75	125.2	**125.2**	0.00	**125.2**	0.00	**125.2**	0.08
S_R4	25	69	10	25	157.6	**157.6**	0.00	**157.6**	0.00	**157.6**	0.08
S_R5	25	69	10	50	272.2	**272.2**	0.00	**272.2**	0.00	**272.2**	0.08
S_R6	25	69	10	75	409.4	**409.4**	0.00	**409.4**	0.00	**409.4**	0.09
S_R7	25	204	10	25	273.4	**273.4**	0.02	**273.4**	0.00	**273.4**	0.09
S_R8	25	204	10	50	507.0	**507.0**	0.01	**507.0**	0.00	**507.0**	0.09
S_R9	25	204	10	75	785.8	**785.8**	0.01	**785.8**	0.00	**785.8**	0.09
S_R10	50	85	10	25	174.6	175.4	0.03	176.0	0.03	**174.6**	0.20
S_R11	50	85	10	50	280.8	**280.8**	0.03	281.6	0.03	**280.8**	0.20
S_R12	50	85	10	75	348.0	**348.0**	0.02	349.2	0.03	**348.0**	0.23
S_R13	50	232	10	25	386.2	389.4	0.07	386.8	0.05	**386.2**	0.26
S_R14	50	232	10	50	708.6	**708.6**	0.06	**708.6**	0.05	**708.6**	0.28
S_R15	50	232	10	75	951.6	**951.6**	0.04	**951.6**	0.05	**951.6**	0.28
S_R16	50	784	10	25	602.0	602.2	0.11	**602.0**	0.05	**602.0**	0.35
S_R17	50	784	10	50	1171.8	1172.2	0.15	1172.0	0.05	**1171.8**	0.42
S_R18	50	784	10	75	1648.8	1649.4	0.14	**1648.8**	0.05	**1648.8**	0.37
S_R19	75	157	10	25	318.2	321.0	0.13	320.0	0.11	**318.2**	0.54
S_R20	75	157	10	50	521.6	526.2	0.14	525.0	0.12	522.6	0.53
S_R21	75	157	10	75	751.0	757.2	0.11	754.2	0.12	**751.0**	0.53
S_R22	75	490	10	25	635.8	638.6	0.16	**635.8**	0.18	**635.8**	0.60
S_R23	75	490	10	50	1226.6	1230.6	0.27	1228.6	0.19	1227.6	0.76
S_R24	75	490	10	75	1789.4	1793.6	0.13	**1789.4**	0.20	**1789.4**	0.77
S_R25	75	1739	10	25	889.8	891.0	0.40	**889.8**	0.16	**889.8**	0.91
S_R26	75	1739	10	50	1664.2	1664.8	0.35	**1664.2**	0.15	**1664.2**	0.86
S_R27	75	1739	10	75	2452.2	2452.8	0.33	**2452.2**	0.16	**2452.2**	0.84
agv agv$_t$						**1.19**	**0.10**	**0.54**	**0.07**	**0.07**	**0.36**
Hits						**13/27**		**18/27**		**25/27**	

Table 1: Test results carried out on the small instances of random graphs.

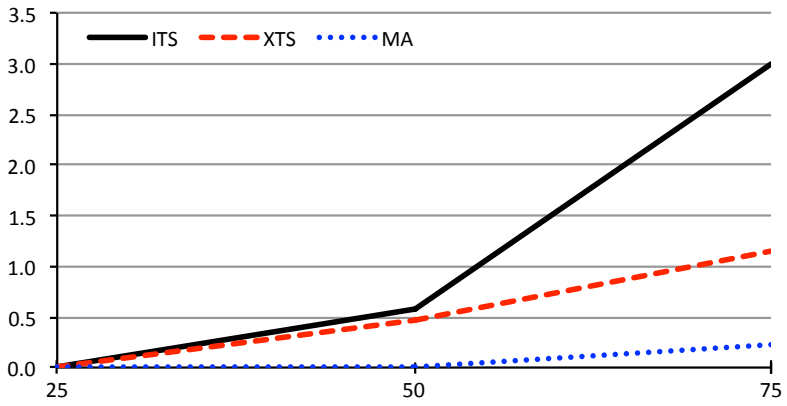

Figure 4: The average gap value of the three algorithms, on the small random graphs, with respect to the instance size.

column as the baseline, e.g. the agv of ITS algorithm is calculated by: $\left(\sum_{i=1}^{p} ITS_i - \sum_{i=1}^{p} Opt_i\right)/p$ where Opt_i and ITS_i are the optimal value and the solution value, computed by ITS, on the i-th instance, respectively, and p is the number of instances in the table. The last line (Hits) of the table shows how many times the algorithms find the optimal solution.

Random Graph. On the instances with 25 nodes the optimal solution is always found by the three algorithms. On size 50, the optimal solution is found 4 times out of 9 by ITS and by XTS and it is always found by MA. On size 75, MA does not find the optimal solution two times (S_R20 and S_R23) whereas XTS misses it 4 times and ITS never finds it. In the following, the terms "hit" and "miss", are used to denote if an algorithm finds or not the optimal solution, on the small instances, or the best solution, on the large instances, respectively. On the 27 instances of Table 1, 13 hits for ITS, 18 hits for XTS and 25 hits for MA occur with an agv equal to 1.19, 0.54 and 0.07, respectively. These results show that MA overcomes the effectiveness of the other two algorithms on random instances.

From the results of Table 1 it is possible to find the characteristics of the instances that affect the effectiveness of the algorithms. For instance, ITS never finds the optimal solution on instances with range weight 10-25, when $n = 50$ and $n = 75$. Moreover, the 0 hits of ITS, verified on the instances with 75 vertices, show that this algorithm loses effectiveness, as the size of problem increases. For XTS the worst results (0 hits) are obtained on the graphs with lowest density (S_R10, S_R11, S_R12, S_R19, S_R20, S_R21). Unlike the previous two metaheuristics, neither the size nor the weight range nor the density seem to affect the effectiveness of MA as shown by its 25 hits out of 27 instances.

In order to show the agv trend of the three algorithms, as the size of the problems increases, we introduce another value (agv_s) representing the average gap value but computed on the instances with the same size. For instance, on the small random graphs, the agv_s of ITS are equal to 0.00, to 0.58 and to 3.00 for n=25, n=50 and n=75, respectively. The agv_s of the three algorithms is plotted in Figure 4 where on the x-axis the size of the instances is reported and on the y-axis the corresponding agv_s value. This figure clearly shows that the agv_s of ITS increases rapidly with the instance size while the growth of agv_s for the other two algorithms is slower. It is interesting to notice that, on size 75, the agv_s of MA is five times lower than agv_s of XTS and thirteen times lower than agv_s of ITS. Regarding the performance, all the algorithms are very fast with a running time that is negligible because always lower than one second.

Squared and Not Squared Grid Graphs. The Tables 2 and 3 report the results on the squared and not squared grid graphs, respectively. Here the columns x and y represent the number of the rows and of the columns of the grid, respectively. From results of Tables 2 it is evident that ITS is the least effective algorithm with only 2 hits (S_SG1 and S_SG2) out of 9 instances and an agv that is the highest one (1.73). Since ITS has a similar behaviour also on the remaining classes of graphs, in the following the comparison

Squared Grid graphs: Small Instances												
Id	**Instance**				**Opt**	**ITS**		**XTS**		**MA**		
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
S_SG1	5	5	10	25	114.0	**114.0**	0.00	**114.0**	0.00	**114.0**	0.08
S_SG2	5	5	10	50	199.8	**199.8**	0.00	**199.8**	0.00	**199.8**	0.08
S_SG3	5	5	10	75	312.4	312.6	0.00	**312.4**	0.00	**312.4**	0.08
S_SG4	7	7	10	25	252.0	252.4	0.03	**252.0**	0.03	**252.0**	0.18
S_SG5	7	7	10	50	437.6	439.8	0.03	**437.6**	0.02	**437.6**	0.18
S_SG6	7	7	10	75	713.6	718.4	0.03	717.4	0.02	**713.6**	0.18
S_SG7	9	9	10	25	442.2	444.2	0.22	442.8	0.13	**442.2**	0.52
S_SG8	9	9	10	50	752.2	754.6	0.29	753.0	0.14	**752.2**	0.59
S_SG9	9	9	10	75	1134.4	1138.0	0.13	**1134.4**	0.14	**1134.4**	0.51
agv agv$_t$						**1.73**	**0.08**	**0.58**	**0.05**	**0.00**	**0.27**
Hits						**2/9**		**6/9**		**9/9**	

Table 2: Test results on the small squared grid graphs.

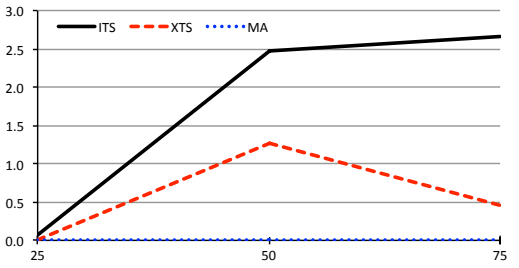

Figure 5: The agv_s on the small squared grid graphs.

Not Squared Grid graphs: Small Instances												
Id	**Instance**				**Opt**	**ITS**		**XTS**		**MA**		
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
S_NG1	8	3	10	25	96.8	**96.8**	0.00	**96.8**	0.00	**96.8**	0.07
S_NG2	8	3	10	50	157.4	**157.4**	0.00	**157.4**	0.00	**157.4**	0.07
S_NG3	8	3	10	75	220.0	**220.0**	0.00	**220.0**	0.00	**220.0**	0.08
S_NG4	9	6	10	25	295.6	295.8	0.07	295.8	0.04	**295.6**	0.29
S_NG5	9	6	10	50	488.6	489.4	0.04	**488.6**	0.04	**488.6**	0.23
S_NG6	9	6	10	75	755.0	**755.0**	0.04	755.2	0.04	**755.0**	0.26
S_NG7	12	6	10	25	398.2	399.8	0.15	398.8	0.09	398.4	0.34
S_NG8	12	6	10	50	671.8	673.4	0.12	**671.8**	0.09	**671.8**	0.36
S_NG9	12	6	10	75	1015.2	1017.4	0.10	1015.4	0.09	**1015.2**	0.42
agv agv$_t$						**0.71**	**0.06**	**0.13**	**0.04**	**0.02**	**0.24**
Hits						**4/9**		**5/9**		**8/9**	

Table 3: Test results on the small not squared grid graphs.

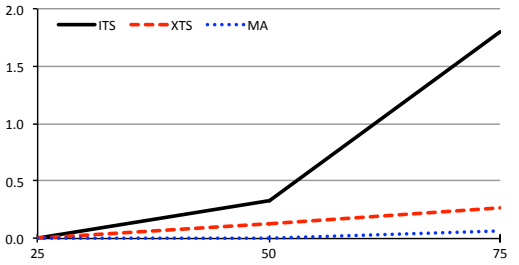

Figure 6: The agv_s on the small not squared grid graphs.

will be more focused on the other two algorithms. XTS finds 6 times the optimal solution, with an *agv* equal to 0.58, whereas MA does even better with 9 hits out of 9 instances. On the not squared grid graphs there is an improvement of ITS results with 4 hits and an *agv* decreased to 0.71. Lightly better are the results of XTS that reports 5 hits with an *agv* equal to 0.13. Here it is evident how the range weight affect the effectiveness of XTS because all its misses always and only occur on the instances with range weight 10-25 and 10-75. The best results are obtained by MA with 8 hits and an *agv* equals to 0.02. Also on the grid graphs the running time of the three algorithm is still negligible.

The graphics depicted in Figures 5 and 6 show a very slow increase of *agv* for MA, as the problem size increases. Since similar results are shown by MA also on the random graphs (Figure 4), we derive that the effectiveness of our algorithm is affected only by problem size. The situation is completely different for the other two algorithms that show less stable results. Indeed, on the squared grid graphs with size 50, the agv_s of XTS is greater than 1 and for ITS is around 2.5 while on the not squared grid graphs with same size both values are lower than 0.5. This shows that the class of graphs used influences the effectiveness of ITS and XTS algorithms.

Toroidal and Hypercube Graphs. The last set of the small instances concerns the toroidal and hypercube graphs which results are reported into Tables 4 and 5, respectively. On the toroidal graphs the number

Toroidal graphs: Small Instances												
Id	**Instance**				**Opt**	**ITS**		**XTS**		**MA**	
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
S_T1	5	5	10	25	101.4	**101.4**	0.00	**101.4**	0.00	**101.4**	0.08
S_T2	5	5	10	50	124.4	**124.4**	0.00	**124.4**	0.00	**124.4**	0.08
S_T3	5	5	10	75	157.8	**157.8**	0.00	158.8	0.00	**157.8**	0.08
S_T4	7	7	10	25	195.4	197.4	0.03	**195.4**	0.03	**195.4**	0.23
S_T5	7	7	10	50	234.2	**234.2**	0.02	**234.2**	0.03	**234.2**	0.20
S_T6	7	7	10	75	269.6	**269.6**	0.02	**269.6**	0.03	**269.6**	0.22
S_T7	9	9	10	25	309.6	310.4	0.20	309.8	0.14	309.8	0.51
S_T8	9	9	10	50	369.6	370.0	0.17	**369.6**	0.15	**369.6**	0.52
S_T9	9	9	10	75	431.8	432.2	0.16	432.2	0.15	**431.8**	0.56
agv agv$_t$						**0.40**	**0.07**	**0.18**	**0.06**	**0.02**	**0.28**
Hits						**5/9**		**6/9**		**8/9**	

Table 4: Test results on the small toroidal graphs.

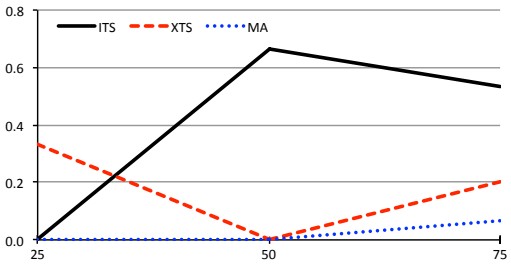

Figure 7: The agv_s on the small toroidal graphs.

Hypercube graphs: Small Instances											
Id	**Instance**			**Opt**	**ITS**		**XTS**		**MA**	
	n	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
S_H1	16	10	25	72.2	**72.2**	0.00	**72.2**	0.00	**72.2**	0.05
S_H2	16	10	50	93.8	**93.8**	0.00	**93.8**	0.00	**93.8**	0.05
S_H3	16	10	75	97.4	**97.4**	0.00	**97.4**	0.00	**97.4**	0.05
S_H4	32	10	25	170.0	**170.0**	0.01	**170.0**	0.01	**170.0**	0.10
S_H5	32	10	50	240.6	241.0	0.00	**240.6**	0.01	**240.6**	0.11
S_H6	32	10	75	277.6	**277.6**	0.00	**277.6**	0.01	**277.6**	0.11
S_H7	64	10	25	353.4	354.6	0.13	353.8	0.08	353.8	0.30
S_H8	64	10	50	475.6	476.0	0.05	**475.6**	0.09	**475.6**	0.35
S_H9	64	10	75	503.8	**503.8**	0.05	504.8	0.09	**503.8**	0.38
agv agv$_t$					**0.22**	**0.03**	**0.16**	**0.03**	**0.04**	**0.17**
Hits					**6/9**		**7/9**		**8/9**	

Table 5: Test results on the small hypercube graphs.

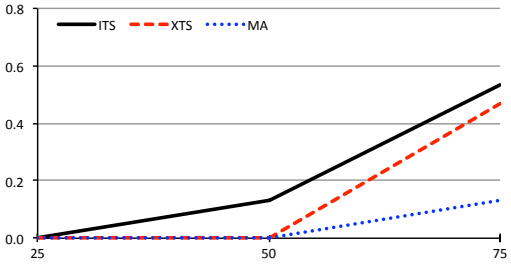

Figure 8: The agv_s on the small hypercube graphs.

of hits is equal to 5 for ITS, 6 for XTS and 8 for MA with an agv equal to 0.40, 0.18 and 0.02, respectively. The results on the single instances show a behaviour of the three algorithms already verified on the other tables. ITS never finds the optimal solution on the largest instances with 81 vertices. On the same instances, the misses of XTS occur when the range weight is 10-25 or 10-75, as already verified on the not squared grid graphs. A single miss (S_T7) occurs for the MA but with a very low gap equal to 0.06.

Finally, on the hypercube graph we have 6 hits for ITS, 7 hits for XTS and 8 hits for MA with an agv equal to 0.22, 0.16 and 0.04, respectively. According to the number of the optimal solutions found and to the very low agv, the hypercube graphs appear to be the easiest instances to solve by the three algorithms. We will see later that this situation changes radically on the large instances.

From the results shown in the previous tables, it is evident that there are several instances where either ITS or XTS algorithms do not find the optimal solution. However, these failures are not so regular as on instances S_R10, S_R19 of Table 1 and the 7^{th} instance of the remaining four tables where these two algorithms never find the optimal solution. For this reason, we focus our attention on these instances, in order to individuate any property that could justify these results. There are two characteristics that these instances share: the range weight 10-25 and the low density. Our hypothesis is that the range weight 10-25 makes very small the gap between the optimal solution value and the local minimum values. Moreover, the

low density of these graphs makes harder "to jump" from a local minimum to another one. As a result, it is easier for the ITS and XTS to be trapped into a local minimum. The complexity of these instances is further certified by results of MA with its three misses on instances S_NG7, S_T7 and S_H7.

In conclusion, from the results reported in Tables 1-5 it is evident that MA widely overcomes the effectiveness of ITS and of XTS algorithms. Indeed, our algorithm, with only 5 misses out of a total of 63 instances, reaches a success percentage around 92% that is much higher than 67% of XTS (with 21 misses) and than 48% of ITS (with 33 misses). Besides the number of hits, MA shows also a better agv that, in the worst case, is equal to 0.07 (on the random graphs) while this value increases to 0.58 for XTS and to 1.73 for ITS (both on the squared grid graphs). This prove that, every time a misses occurs for MA, the solutions computed by our algorithm are very close to the optimal one. This behaviour is confirmed also by agv_s values. Indeed, the diagrams depicted in Figures 4-8 show that the agv_s of MA is always lower than or equal to the agv_s of the other two algorithms, whatever is the size of instances and the class of graphs used.

Regarding the performance, MA appears lightly slower than the other two algorithms but its computational time is negligible because almost always lower than one second. This makes our algorithm particularly suitable to be embedded into an exact approach where it is necessary to quickly generate upper bounds as tight as possible. Where the instances present many local minimum very close to the optimal solution, it is fundamental the individuation of very tight upper bounds because they can heavily speed up the exact approach. This justifies our goal to improve the previous two metaheuristics although their results were already very close to the optimal solution.

Finally, another very important property of MA, that makes it preferable to the other two algorithms, is its robustness. The effectiveness of MA is essentially affected by instances size, as shown by its misses that always occur on the largest instances. Different is the situation of the other two metaheuristics. The main characteristic that affect the effectiveness of ITS is the size of problem but, unlike MA, its misses often occurs also on instances with only 50 vertices. Moreover, the results, on the random and on the toroidal graphs, show that ITS is not able to find the optimal solution when the range weight is 10-25 and the number of vertices is greater than 25. Finally, for XTS the hardest instances to solve are those with low density and range weight 10-25 and 10-75.

5.2 Large Graphs

To correctly evaluate a metaheuristics it is necessary to verify both its effectiveness and its performance on the small instances, where the comparison is made with the optimal solutions, and on the large instances, where the comparison is made directly with the solutions of the other metaheuristics. On the small instances we showed that MA is the more effective algorithm. Now we have to verify if MA preserves its effectiveness also on the large instances and what is its performance on these instances. To this end, we run the three algorithms on a set of instances with a size that ranges from 100 to 500 vertices.

Random graphs: Large Instances											
Id	**Instance**				**Best**	**ITS**		**XTS**		**MA**	
	n	**m**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
L_R1	100	247	10	25	498.4	501.4	0.33	500.8	0.31	**498.4**	0.93
L_R2	100	247	10	50	836.8	845.8	0.37	840.0	0.34	**836.8**	1.28
L_R3	100	247	10	75	1207.6	1223.8	0.28	1208.0	0.34	**1207.6**	1.20
L_R4	100	841	10	25	826.8	828.2	0.27	**826.8**	0.44	**826.8**	1.08
L_R5	100	841	10	50	1724.4	1729.6	0.60	1724.6	0.45	**1724.4**	1.66
L_R6	100	841	10	75	2420.6	2425.6	0.35	**2420.6**	0.45	**2420.6**	1.45
L_R7	100	3069	10	25	1134.0	**1134.0**	0.59	**1134.0**	0.35	**1134.0**	1.70
L_R8	100	3069	10	50	2179.0	**2179.0**	0.69	**2179.0**	0.36	**2179.0**	1.84
L_R9	100	3069	10	75	3228.6	3228.8	0.77	3228.8	0.34	**3228.6**	2.08
L_R10	200	796	10	25	1468.2	1488.4	3.48	1468.8	3.33	**1468.2**	6.01
L_R11	200	796	10	50	2399.0	2442.6	2.50	2414.4	3.45	**2399.0**	8.20
L_R12	200	796	10	75	3089.6	3157.0	2.78	3099.6	3.45	**3089.6**	10.34
L_R13	200	3184	10	25	1986.2	2003.6	2.78	1986.8	3.42	**1986.2**	10.54
L_R14	200	3184	10	50	3650.6	3683.6	2.67	**3650.6**	3.67	3651.8	10.84
L_R15	200	3184	10	75	5135.8	5158.6	2.76	5137.2	3.56	**5135.8**	13.39
L_R16	200	12139	10	25	2447.8	2450.0	11.31	2448.4	2.81	**2447.8**	26.35
L_R17	200	12139	10	50	4148.6	4149.4	8.91	**4148.6**	2.79	4149.0	20.62
L_R18	200	12139	10	75	5528.4	5531.4	6.98	**5528.4**	2.87	**5528.4**	21.93
L_R19	300	1644	10	25	2045.4	2072.6	10.19	**2045.4**	12.42	2048.0	23.16
L_R20	300	1644	10	50	4175.4	4239.4	9.12	4195.2	12.84	**4175.4**	24.27
L_R21	300	1644	10	75	6065.2	6154.4	11.09	6102.8	12.74	**6065.2**	32.61
L_R22	300	7026	10	25	3203.0	3231.0	19.59	**3203.0**	11.98	3207.6	42.07
L_R23	300	7026	10	50	6211.0	6261.4	21.12	**6211.0**	12.64	6217.2	55.05
L_R24	300	7026	10	75	8585.4	8660.6	17.21	**8585.4**	12.46	8613.2	50.78
L_R25	300	27209	10	25	3726.6	3729.2	44.74	**3726.6**	9.61	**3726.6**	108.05
L_R26	300	27209	10	50	5734.8	5738.0	29.26	**5734.8**	9.55	**5734.8**	97.12
L_R27	300	27209	10	75	10467.0	10469.6	50.88	**10467.0**	9.56	**10467.0**	106.78
L_R28	400	2793	10	25	2989.6	3015.2	29.99	2991.0	31.61	**2989.6**	49.95
L_R29	400	2793	10	50	6410.0	6528.0	35.82	6435.8	32.16	**6410.0**	73.12
L_R30	400	2793	10	75	8597.2	8730.0	35.36	8637.0	32.14	**8597.2**	75.73
L_R31	400	12369	10	25	4428.8	4451.8	55.14	**4428.8**	28.92	4437.4	142.79
L_R32	400	12369	10	50	6785.8	6837.4	35.88	**6785.8**	28.97	6800.6	141.25
L_R33	400	12369	10	75	10599.4	10661.8	48.12	**10599.4**	28.40	10601.0	191.50
L_R34	400	48279	10	25	5060.4	5060.8	123.27	**5060.4**	23.65	5060.6	282.71
L_R35	400	48279	10	50	7106.8	7109.2	85.15	**7106.8**	23.57	7108.0	265.29
L_R36	400	48279	10	75	15103.2	15114.6	127.31	**15103.2**	24.02	15117.8	240.77
L_R37	500	4241	10	25	4056.4	4102.8	68.35	4063.0	64.11	**4056.4**	97.47
L_R38	500	4241	10	50	7170.4	7285.0	70.14	7204.6	64.40	**7170.4**	127.22
L_R39	500	4241	10	75	11135.6	11285.6	63.93	11179.6	63.65	**11135.6**	163.00
L_R40	500	19211	10	25	5724.2	5745.8	99.12	**5724.2**	56.97	5741.4	520.63
L_R41	500	19211	10	50	7677.8	7725.0	89.63	**7677.8**	55.49	7678.2	537.04
L_R42	500	19211	10	75	14124.8	14167.8	80.09	**14124.8**	56.41	14164.8	546.12
L_R43	500	75349	10	25	6361.6	6366.4	181.71	6362.0	47.09	**6361.6**	489.38
L_R44	500	75349	10	50	8668.4	8671.2	155.18	**8668.4**	48.48	**8668.4**	536.60
L_R45	500	75349	10	75	16932.4	16939.2	201.96	**16932.4**	49.03	16933.8	498.47
agv agv$_t$						**32.39**	**41.06**	**5.44**	**19.90**	**3.17**	**125.79**
Hits						**2/45**		**25/45**		**29/45**	

Table 6: Test results on the large instances of random graphs.

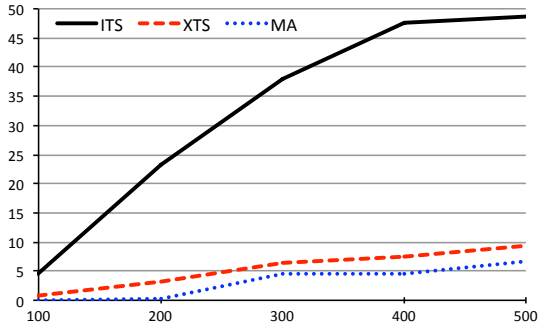

Figure 9: The agv_s on the large random graphs.

Random Graphs. Table 6 reports the results on the large random graphs. The structure of the table is similar to that of Table 1, except for the column *Opt* that is replaced by the column *Best* that reports the best solution (the optimal one is not known for these instances), i.e. the minimum one, among those computed by the three metaheuristics. In this case, the last line of the table shows how many times each algorithm finds the best solution. There are only 2 hits for ITS, on the instances with 100 vertices (L_R7 and L_R8) and its agv is high (32.39). These results reveal the difficulties of ITS as the size of the problem increases, indeed no hits occur on instances with at least 200 vertices. Because of these results the remaining evaluations, on the large random instances, are carried out between XTS and MA.

The results of MA are better than the results of XTS, with 29 hits instead of 25. Moreover, the agv of MA (3.17) is lower than the agv of XTS (5.44). Also on the large instances, the difficulties of XTS are evident on the instances with low density (L_1-3, L_10-12,...). Indeed, on these instances the best solution is computed always by MA except for the instance L_R19. The opposite situation occurs on the graphs with medium density (L_R4-6,L_R13-15,...) and with at least 300 vertices on whose the best solution is always computed by XTS. We observe here a reduction of MA effectiveness as the size and density of instances increase. This behaviour was expected and it is due to a lower number of iterations carried out by our algorithm. Recall that, according to the formula that computes $MaxIt$, as the density and the size increase, $MaxIt$ decreases. This policy is necessary to obtain a trade-off between the effectiveness and performance of MA because the local search procedures are more expensive on dense graphs. Indeed, from the value of avg_t it is evident that MA is much slower than the other two algorithms. Our algorithm runs in less than 5 minutes, on instances up to 400 vertices, and in less than 10 minutes, on instances with 500 vertices. We consider acceptable this last running time because it occurs only on instances with 500 vertices and it allows MA to compute the best solution in 65% of cases that is better than 55% obtained by XTS. Moreover, the agv_s of MA is always lower than agv_s of XTS, as shown in Figure 9. The diagram depicted in this figure shows that the quality of the solutions produced by MA is better than the solutions of XTS, whatever are the size of the problem and the number of misses occurred. Finally, as shown in the follows, the random graphs represent the pathological case for the performance of MA because, on the other classes of graphs, its performance is much better.

Squared and Not Squared Grid Graphs. Tables 7 and 8 report the results on large squared and not squared grid graphs. From these results it is evident that the effectiveness of ITS and, specially, of XTS is strongly reduced on these classes of graphs. On the squared grid graphs ITS never finds the best solution, it has the highest agv (14.60) and it is also the slowest algorithm. There are only 3 hits for XTS with an agv equal to 6.37. This is a considerable reduction of XTS effectiveness that passes from $\sim 55\%$ of hits, on the large random graphs, to the $\sim 20\%$ on the large squared grid graphs. The best results are produced by MA with 12 hits and an agv equals to 0.87. Figure 10 shows that also the agv_s of MA is always lower than or equal to the agv_s of the other two algorithms. In particular, the results, on the instances with 500 vertices,

Squared Grid graphs: Large Instances												
Id	**Instance**				**Best**	**ITS**		**XTS**		**MA**		
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
L_SG1	10	10	10	25	566.8	570.6	0.54	**566.8**	0.26	567.0	0.69
L_SG2	10	10	10	50	947.0	948.8	0.41	949.4	0.26	**947.0**	0.88
L_SG3	10	10	10	75	1557.8	1566.0	0.51	1565.2	0.27	**1557.8**	0.84
L_SG4	14	14	10	25	1206.4	1209.4	8.07	1207.6	2.40	**1206.4**	3.43
L_SG5	14	14	10	50	2007.8	2008.6	8.06	2010.2	2.46	**2007.8**	3.90
L_SG6	14	14	10	75	3399.2	3401.2	7.23	3406.0	2.42	**3399.2**	5.06
L_SG7	17	17	10	25	1829.4	1834.2	42.63	1830.4	9.20	**1829.4**	10.95
L_SG8	17	17	10	50	3051.4	3070.6	29.71	3062.8	9.23	**3051.4**	10.65
L_SG9	17	17	10	75	5071.2	5089.8	29.68	**5071.2**	9.32	5072.2	16.69
L_SG10	20	20	10	25	2602.2	2619.8	85.42	2607.8	28.64	**2602.2**	19.35
L_SG11	20	20	10	50	4299.8	4321.2	103.84	4306.6	29.00	**4299.8**	25.07
L_SG12	20	20	10	75	7240.4	7272.6	127.81	**7240.4**	28.95	7252.2	26.50
L_SG13	23	23	10	25	3453.6	3462.8	371.23	3460.6	78.43	**3453.6**	45.78
L_SG14	23	23	10	50	5831.4	5865.4	291.52	5837.4	80.03	**5831.4**	55.77
L_SG15	23	23	10	75	9681.0	9723.4	240.50	9718.6	79.80	**9681.0**	69.33
agv agv$_t$						**14.60**	**89.81**	**6.37**	**24.04**	**0.87**	**19.66**
Hits						**0/15**		**3/15**		**12/15**	

Table 7: Test results on the large squared grid graphs.

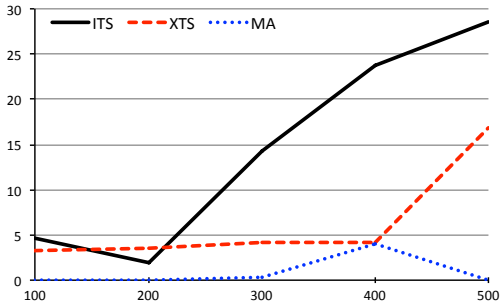

Figure 10: The agv_s on the large squared grid graphs.

Not Squared Grid graphs: Large Instances												
Id	**Instance**				**Best**	**ITS**		**XTS**		**MA**		
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
L_NG1	13	7	10	25	512.20	513.00	0.36	513.00	0.19	**512.20**	0.76
L_NG2	13	7	10	50	803.40	**803.40**	0.31	**803.40**	0.20	**803.40**	0.62
L_NG3	13	7	10	75	1382.80	1390.80	0.34	1386.00	0.20	**1382.80**	1.06
L_NG4	18	11	10	25	1204.60	1208.00	6.78	1206.40	2.46	**1204.60**	5.23
L_NG5	18	11	10	50	2041.20	2049.80	8.77	2047.60	2.48	**2041.20**	4.52
L_NG6	18	11	10	75	3417.40	3431.00	5.79	3422.20	2.48	**3417.40**	4.53
L_NG7	23	13	10	25	1923.80	1930.60	42.54	**1923.80**	10.25	1925.20	14.92
L_NG8	23	13	10	50	3178.20	3194.80	43.01	3187.60	10.43	**3178.20**	13.74
L_NG9	23	13	10	75	5258.00	5286.60	34.27	5286.00	10.40	**5258.00**	18.73
L_NG10	26	15	10	25	2522.60	2532.80	104.81	**2522.60**	25.75	2529.40	25.86
L_NG11	26	15	10	50	4144.20	4164.80	82.30	4148.40	25.87	**4144.20**	22.17
L_NG12	26	15	10	75	7031.40	7063.40	85.79	**7031.40**	25.83	7035.80	21.92
L_NG13	29	17	10	25	3255.00	3270.00	236.94	3257.00	59.92	**3255.00**	33.89
L_NG14	29	17	10	50	5410.80	5430.40	251.17	5422.60	60.35	**5410.80**	44.42
L_NG15	29	17	10	75	8953.00	8993.20	196.66	8985.60	60.37	**8953.00**	47.74
agv agv$_t$						**14.93**	**73.32**	**7.00**	**19.81**	**0.84**	**17.34**
Hits						**1/15**		**4/15**		**12/15**	

Table 8: Test results on the large not squared grid graphs.

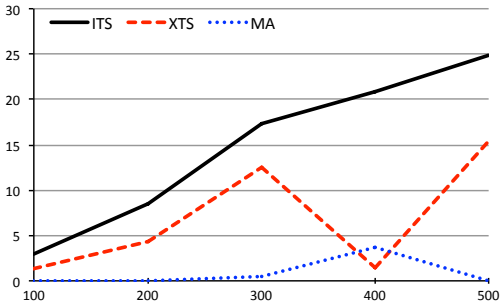

Figure 11: The agv_s on the large not squared grid graphs.

show that the agv_s of MA is 15 and 25 units lower than the agv_s of XTS and of ITS, respectively. It is important here to highlight that all these results are obtained by MA with an avg_t equal to 19.66 seconds that is lower than 24.04 seconds of XTS and much lower than 89.81 seconds of ITS. Very similar results are observed on the not squared grid graphs on whose there are two aspects to highlight: MA is again the fastest algorithm and, for the first time, the agv_s of MA is greater than the agv_s of XTS, on the instances with 400 vertices, as shown in Figure 11.

Toroidal and Hypercube Graphs. In Table 9 and Table 10 the results on the toroidal and hypercube graphs are shown. On the toroidal graphs there are 0 hits for ITS and 3 hits for XTS with an agv equal to 4.65 and 2.20, respectively. On the contrary, there is only a miss for MA (L_T4) that always finds the best solutions on the instances with at least 289 vertices. Moreover, the agv equal to 0.05 is the lowest one obtained by MA on the whole set of largest instances. The Figure 12 shows that, on the instances with 500 vertices, ITS and XTS have a similar agv_s that is around 7 units greater than the agv_s of MA. Finally, for

Toroidal graphs: Large Instances											
Id	**Instance**				**Best**	**ITS**		**XTS**		**MA**	
	x	**y**	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
L_T1	10	10	10	25	388.0	388.8	0.38	389.0	0.28	**388.0**	0.80
L_T2	10	10	10	50	457.6	458.6	0.37	**457.6**	0.29	**457.6**	0.72
L_T3	10	10	10	75	504.6	504.8	0.25	**504.6**	0.29	**504.6**	0.75
L_T4	14	14	10	25	748.8	750.8	5.96	**748.8**	2.43	749.6	4.14
L_T5	14	14	10	50	874.4	875.6	3.68	875.4	2.49	**874.4**	3.52
L_T6	14	14	10	75	1016.2	1017.2	3.59	1016.4	2.51	**1016.2**	3.85
L_T7	17	17	10	25	1102.8	1110.2	21.98	1107.4	9.07	**1102.8**	7.01
L_T8	17	17	10	50	1304.4	1307.6	20.93	1306.0	9.21	**1304.4**	8.48
L_T9	17	17	10	75	1498.6	1502.4	23.18	1499.6	9.25	**1498.6**	12.00
L_T10	20	20	10	25	1539.6	1548.6	88.75	1540.0	28.31	**1539.6**	18.01
L_T11	20	20	10	50	1795.4	1803.4	81.03	1797.6	28.83	**1795.4**	19.77
L_T12	20	20	10	75	2033.0	2042.6	55.19	2033.6	28.78	**2033.0**	19.45
L_T13	23	23	10	25	2034.8	2043.4	278.08	2043.8	77.39	**2034.8**	50.01
L_T14	23	23	10	50	2406.4	2412.2	177.53	2410.8	79.20	**2406.4**	40.77
L_T15	23	23	10	75	2697.2	2705.4	184.99	2704.2	78.64	**2697.2**	43.95
agv agv$_t$						**4.65**	**63.06**	**2.20**	**23.80**	**0.05**	**15.55**
Hits						**0/15**		**3/15**		**14/15**	

Table 9: Test results on the large toroidal graphs.

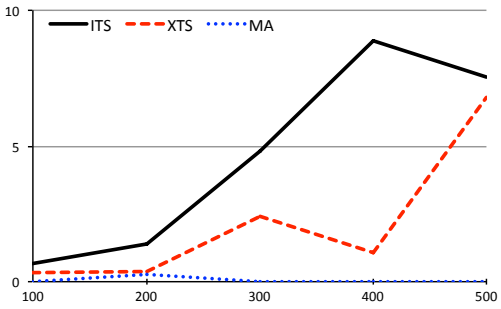

Figure 12: The agv_s on the large toroidal graphs.

Hypercube graphs: Large Instances										
Id	**Instance**			**Best**	**ITS**		**XTS**		**MA**	
	n	**low**	**up**		**Value**	**Time**	**Value**	**Time**	**Value**	**Time**
L_H1	128	10	25	731.8	740.0	1.09	742.0	0.71	**731.8**	1.11
L_H2	128	10	50	1066.8	1071.0	0.40	**1066.8**	0.76	1067.2	1.75
L_H3	128	10	75	1161.6	1163.6	0.34	**1161.6**	0.77	1162.4	1.98
L_H4	256	10	25	1487.4	1542.6	9.41	1534.2	6.58	**1487.4**	5.03
L_H5	256	10	50	2279.6	2311.4	6.45	2282.0	6.80	**2279.6**	11.54
L_H6	256	10	75	2572.4	2590.8	3.94	2576.4	6.85	**2572.4**	13.23
L_H7	512	10	25	3119.0	3240.8	73.51	3146.0	63.07	**3119.0**	49.83
L_H8	512	10	50	4852.2	4921.8	67.58	4872.4	66.43	**4852.2**	77.71
L_H9	512	10	75	5553.4	5588.6	51.74	5563.8	67.65	**5553.4**	85.99
agv agv$_t$					**38.49**	**23.80**	**13.44**	**24.40**	**0.13**	**27.60**
Hits					**0/9**		**2/9**		**7/9**	

Table 10: Test results on the large hypercube graphs.

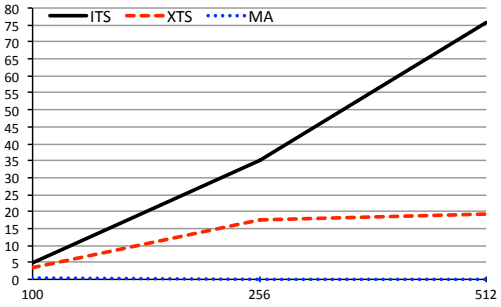

Figure 13: The agv_s on the large hypercube graphs.

the third time our algorithm is the fastest one.

The results of Table 10 reveal the great difficulties of XTS and of ITS algorithms on the hypercube graphs. XTS finds the best solutions only in two cases (L_H2 and L_H3) on the instances with only 128 vertices and its agv, equal to 13.44, is the highest agv of XTS on the whole set of large instances. Moreover, Figure 13 shows that the agv_s of XTS is 20 units greater than the agv_s of MA on sizes 256 and 512. Even worse are the results of ITS that never finds the best solution and it obtains its worst agv (38.49). Moreover, on the instances with 512 vertices its agv_s is 75 units greater than the agv_s of MA. On the contrary, very good are the results of MA with 7 hits, an agv equal to 0.13 and an agv_s equal to 0 for the instances with at least 256 vertices. These results are obtained with a computational time that is greater than the other two algorithms but of only 3 seconds.

In conclusion, the results obtained on the large instances shows that MA is the more effective algorithm with a percentage of hits equal to 70%. This last value is much higher than 3% and 35% obtained by ITS and XTS, respectively. Moreover, on the instances up to 300 vertices, the percentage of hits of MA grows

up to 80% while this percentage decreases on greater instances because MA is forced to use less iterations. Beside the number of hits, it is interesting to evaluate even the quality of the solutions produced by the algorithms when the misses occur. To this end, we analyze the agv and the agv_s of the algorithms whose values prove the robustness of MA. By selecting the best and worst cases, we observe that the agv of ITS ranges from 4.65, on the toroidal graphs, to 38.49, on the hypercube graphs, and the agv of XTS ranges from 2.20, on the toroidal graphs, to 13.44, on the hypercube graphs. These values reveal that the quality of produced solutions depends on the classes of graphs used and, in particular for ITS, the gap from the best solution is relevant. On the contrary, the agv of MA ranges from 0.05, on the toroidal graphs, to 3.17, on the random graphs and this means that, when the misses occur, the solutions found by MA are very close to the best ones, whatever are the classes of graphs. This statement is further highlight by the agv_s values of MA, depicted in Figures 9-13, that show a more regular trend, as the size increases, in respect to the other two algorithms. Regarding the performance, MA is the fastest algorithm on the squared grid, not squared grid and toroidal graphs while it is slower than the other two algorithms on the hypercube graphs and it is significantly slower on the random graphs. The running times of MA on the random graphs highlight that the main parameter that affects the performance of our algorithm is the density of the graph. This is evident by observing that MA requires more time to solve the dense instances L_R34, L_R35, L_R36 composed by 400 nodes than the sparse instances L_R37, L_R38, L_R39 composed by 500 vertices.
Finally, we analyzed also the performance gap, between XTS and MA, taking in account only the instances where the solutions of XTS are equal to or better than the solutions of MA. On the large Random graphs, XTS finds the best solution 25 times and, on these instances, it results up to ten times faster than MA. The situation change radically, on the 54 instances of the other kinds of graphs, with a significant reduction of the performance gap and of the hits (only 12). In particular, on the instances L_SG1, L_NG2, L_NG10, L_T2, L_T3, L_H2 and L_H3 the performance gap is lower than a second. On the instances L_SG9, L_NG7 and LT_4 this gap is around 7, 5 and 2 seconds while on the instances L_SG12 and L_NG12, XTS is slower than MA.

6. Conclusion

In this paper we presented a very effective memetic approach for the WFVS problem. Our algorithm is characterized by three main aspects. The first is the Snd procedure that allows the creation of a variegated initial population that befits to a better exploration of space solutions. The second aspect is the use of two local search procedures, based on k-diamonds. These procedures represent the key of MA effectiveness but they can slow down the algorithm, when invoked too many times. Thank to a wise application of these local search procedure, we obtained a good trade-off between the effectiveness and the performance. The third aspect is the diversification schema that, by applying the penalties on the vertices of incumbent solution, moves the whole population toward a new part of the space solutions.

The computational results show that our algorithm widely overcomes the effectiveness of the other two metaheuristics, proposed in the literature, with a percentage of hits equal to 92%, on the small instances, and equal to 70% on the large instances. Moreover, except for only one case, the *agv* of MA is always lower than the *agv* of ITS and XTS on both the small and the large instances. These results certify the robustness of our algorithm whose effectiveness seems not be affected by the density, by the range weight and by the classes of graphs used. Regarding the performance, on the small instances, the running time of MA is negligible because it is always lower than 1 second (often half a second). This makes our algorithm particularly suitable to be embedded into an exact approach to quickly generate tight upper bounds. On the large instances, MA results the fastest algorithm on three classes of graphs, (squared and not squared grid and toroidal graphs) while it is significantly slower than the other two algorithms on the random graphs. The results on these graphs show that, as the density increases the performance of our algorithm decreases. Currently, this seems the only drawback our MA that essentially depends by the running time of local search procedure that increases as the size of k-diamonds increases. Since the memetic algorithms are fit for parallel implementation, this could be a possible direction for a future work that solves the only drawback of MA.

7. Acknowledgment

The authors wish to thank Marco Trubian who provided the XTS heuristic code. The authors also thank the three anonymous referees for their valuable comments.

References

[1] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics*, 12(3):289–297, 1999.

[2] R. Bar-Yehuda, D. Geiger, J. Naor, and R.M. Roth. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM Journal on Computing*, 27(4):942–959, 1998.

[3] A. Becker and D. Geiger. Approximation algorithms for the loop cutset problem. In *Journal of Artificial Intelligence Research*, pages 60–68. Morgan Kaufmann, 1994.

[4] L. Brunetta, F. Maffioli, and M. Trubian. Solving the feedback vertex set problem on undirected graphs. *Discrete Applied Mathematics*, 101:37–51, 1999.

[5] F. Carrabs, R. Cerulli, M. Gentili, and G. Parlato. Minimum weighted feedback vertex set on diamonds. *Electronic Notes in Discrete Mathematics*, 17:87–91, 2004.

[6] F. Carrabs, R. Cerulli, M. Gentili, and G. Parlato. A linear time algorithm for the minimum weighted feedback vertex set on diamonds. *Information Processing Letters*, 94(1):29–35, 2005.

[7] F. Carrabs, R. Cerulli, M. Gentili, and G. Parlato. A tabu search heuristic based on k-diamonds for the weighted feedback vertex set problem. *Lecture Notes in Computer Science*, 6701:589–602, 2011.

[8] M. Dell'Amico, A. Lodi, and F. Maffioli. Solution of the cumulative assignment problem with a well-structured tabu search method. *Journal of Heuristics*, 5:123–143, 1999.

[9] G. Even, J. Naor, B. Schieber, and L. Zosin. Approximating minimum subset feedback sets in undirected graphs with applications. *SIAM Journal on Discrete Mathematics*, 13(2):255–267, 2000.

[10] Paola Festa, Panos M. Pardalos, and Mauricio G.C. Resende. Feedback set problems. In *Encyclopedia of Optimization*. Springer US, 2009.

[11] R. Focardi, F.L. Luccio, and D.Peleg. Feedback vertex set in hypercubes. *Information Processing Letters*, 76(1-2):1–5, 2000.

[12] P. Galinier, E. Lemamou, and M.W. Bouzidi. Applying local search to the feedback vertex set problem. *Journal of Heuristics*, pages 1–22, 2013.

[13] W. Jiang, T. Liu, C. Wang, and K. Xu. Feedback vertex sets on restricted bipartite graphs. *To appear on Theoretical Computer Science*, 2013.

[14] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[15] J. Kleinberg and A. Kumar. Wavelength conversion in optical networks. *Journal of algorithms*, 38(1):25–50, 2001.

[16] Y. D. Liang. On the feedback vertex set problem in permutation graphs. *Information Processing Letters*, 52:123–129, 1994.

[17] Y.D. Liang and M.S. Chang. Minimum feedback vertex sets in cocomparability graphs and convex bipartite graphs. *Acta Informatica*, 34(5):337–346, 1997.

[18] E. L. Lloyd and M.L. Soffa. On locating minimum feedback vertex sets. *Journal of Computer and System Sciences*, 37(3):292–311, 1988.

[19] C.L. Lu and C.Y. Tang. A linear-time algorithm for the weighted feedback vertex problem on interval graphs. *Information Processing Letters*, 61(2):107–111, 1997.

[20] P. Moscato and C. Cotta. A gentle introduction to memetic algorithms. In F.Glover and G.A.Kochenberger, editors, *Handbook of Metaheuristics*, volume 57, pages 105–144. Springer, 2003.

[21] P.M. Pardalos, T. Qian, and M.G.C. Resende. A greedy randomized adaptive search procedure for the feedback vertex set problem. *Journal of Combinatorial Optimization*, 2:399–412, 1999.

[22] D. Peleg. Local majority voting, small coalitions and controlling monopolies in graphs: a review. In *Proc. of 3rd Colloquium on Structural Information and Communication Complexity*, 1996.

[23] D. Peleg. Size bounds for dynamic monopolies. *Discrete Applied Mathematics*, 86(2–3):263–273, 1998.

[24] A. Shamir. A linear time algorithm for finding minimum cutsets in reduced graphs. *SIAM Journal On Computing*, 8:645–655, 1979.

[25] A. Takaoka, S. Tayu, and S. Ueno. On minimum feedback vertex sets in graphs. In *Proceedings of the 2012 3rd International Conference on Networking and Computing, ICNC 2012*, pages 429–434, 2012.

[26] C.C. Wang, E.L. Lloyd, and M.L. Soffa. Feedback vertex sets and cyclically reducible graphs. *Journal of the Association for Computing Machinery*, 32:296–313, 1985.