A Multi-Ethnic Genetic Approach for the Minimum Conflict Weighted Spanning Tree Problem

Francesco Carrabs*[1], Carmine Cerrone†[2], and Rosa Pentangelo‡[1]

[1]Department of Mathematics, University of Salerno, Fisciano, Italy
[2]Department of Biosciences and Territory, University of Molise, Italy

Abstract

This paper addresses a variant of the minimum spanning tree problem in which, given a list of conflicting edges, the primary goal is to find a spanning tree with the minimum number of conflicting edge pairs and the secondary goal is to minimize the weight of spanning trees without conflicts. The problem is NP-hard and it finds applications in the design of offshore wind farm networks. We propose a multi ethnic genetic algorithm for the problem in which the fitness function is designed to simultaneously manage the two goals of the problem. Moreover, we introduce three local search procedures to improve the solutions inside the population during the computation. Computational results performed on benchmark instances reveal that our algorithm outperforms the other heuristic approach, proposed in the literature, for this problem.

Keywords: Genetic Algorithm; Evolutionary algorithm; Multi-Ethnic; Conflict Edge Pairs; Conflict Constraints; Spanning Tree

*fcarrabs@unisa.it
†carmine.cerrone@unimol.it
‡rpentangelo@unisa.it

1 Introduction

The Minimum Spanning Tree Problem with Conflicting Edge Pairs (MSTC) is a NP-Hard variant of the classical Minimum Spanning Tree problem. Given an undirected and edge weighted graph $G(V, E, P)$, where P is a set of conflicting edge pairs, MSTC problem consists in finding a minimum spanning tree of G without edges in conflict. The problem was introduced by Darmann et al. [7], [8]. In these works, the authors proved that MSTC problem is NP-Hard and that it is polynomially solvable when all the pairs in P are disjointed. Another polynomial case for MSTC problem occurs when the pairs in P satisfy the transitive property ([20]) that is: if $\{e_1, e_2\} \in P$ and $\{e_2, e_3\} \in P$ then even $\{e_1, e_3\}$ is in P.

MSTC problem belongs to the class of NP-Hard variants of the spanning tree problem. Among the others, we find in this class the degree-dependent spanning tree [5], the generalized spanning tree [9] and the spanning tree with minimum branch vertices [2], the bounded degree spanning tree [6].

In the literature, there are several optimization problems with conflict constraints such as the knapsack problem with conflict constraints [15], the maximum flow problems with disjunctive constraints [16], the bin packing problem with conflicts [17] and the minimum cost perfect matching with conflict pair constraints [14].

MSTC problem arises in some real world applications like the design of an offshore wind farm network. The design of such systems is based on the connection layout of wind turbines installed, realized through cables characterized by a certain capacity and a certain cost. Given the capacity of each cable, the system design begins by performing a clustering of the turbines that can be connected to a single cable. Defined a cluster of turbines, the next step consists in connecting them in the cheapest way (thus creating a spanning tree of minimum cost). The additional request is to carry out this connection by avoiding overlapped cables ([12]). By considering two overlapped cables as a conflicting pair, the problem just described coincides with the MSTC. Another application of MSTC concerns the resolution of quadratic bottleneck spanning problem (QBSTP) which is defined as follows. Given a graph G=(V,E), to each couple of edges $(e_1, e_2) \in E \times E$ is associated a weight w_{e_1,e_2}. QBSTP consists in finding a spanning tree T of G such that $\max\{w_{e_1,e_2} : e_1, e_2 \in T\}$ is as small as possible. In [20] the authors show how to use the MSTC to improve the heuristics for the QBSTP. In [11] the authors apply the MSTC on road map where some types of movements are forbidden. For instances, in some point of the map can be forbidden to turn left or right and this constraint can be simulated by using conflict edges. Finally, another application of MSTC arises in the installation of an oil pipeline system connecting various countries [7].

Regarding the MSTC resolution, in [20] the authors proposed several heuristic approaches and two exact algorithms based on Lagrangian relaxation. When a conflict free solution is not found, these heuristics return the number of conflict pairs present in the solutions. In [18] a branch and cut approach were proposed. This algorithm was based on the concept of conflict graph $\hat{G}(E, C)$ in which each edge of the original graph G is mapped in a node of $\hat{G}$ and there is an edge between two vertices in $\hat{G}$ if and only the corresponding edges in G are in conflict. Moreover, the authors introduced a preprocessing phase that results very effective on some sets of instances. Finally, very recently, another branch-and-cut algorithm for MSTC was introduced in [3]. Thanks to a new set of valid inequalities, based on combined properties belonging to any feasible solution, this last algorithm outperforms the ones proposed in [18].

It is worth noting that, by definition, the MSTC problem is infeasible on a graph G when there are not spanning trees without conflicts in G. Rather than marking an instance as infeasible, we prefer to solve a variant of MSTC problem in which the primary goal is to minimize the number of conflict edge pairs in the spanning tree T and, the secondary goal is to minimize the weight of T, if T has no conflicts. This new variant is named Minimum Conflict Weighted Spanning Tree problem (MCWST). It is easy to see that, if exists, the optimal solution of MSTC problem coincides with the optimal solution of MCWST. This means that by solving MCWST we solve the MSTC problem too. In this work, we propose a multi ethnic genetic algorithm for MCWST problem. In particular, we define a fitness function which is able to manage the two goals of the problem at the same time. Moreover, during the computation, we apply three local search procedures to improve the solutions within the population. Finally, we compare the multi-ethnic genetic algorithm with the heuristics proposed in [20] on their benchmark instances.

The rest of the paper is organized as follows. The problem is formally defined in Section 2. The genetic algorithm and the local search procedures are described in Section 3 and Section 4, respectively. The multi ethnic genetic framework is described in Section 5. Finally, the computational results are presented in Section 6 and some concluding remarks are given in Section 7.

2 Notations and problem definition

Let $G(V, E, P)$ be an undirected edge weighted graph, where V is the set of vertices, E the set of edges and $P \subseteq E \times E$ is the set of *conflict edge pairs*. Formally:

$$P = \{\{e_i, e_j\} : e_i \in E, e_j \in E,\ e_i \text{ and } e_j \text{ are in conflict}\}$$

Since the couples in P are not ordered, $\{e_i, e_j\}$ and $\{e_j, e_i\}$ are the same couple. We denote by n and m the cardinality of V and E, respectively, and by w_{e_k} the non negative weight of the edge e_k.

Moreover, $\forall e_k \in E$ let $\mathbb{P}(e_k, E) = \{\{e_k, e_j\} \in P : e_j \in E\}$ be the set of conflict edge pairs containing the edge e_k and $\forall E' \subseteq E$ let $\zeta(E') = \bigcup_{e_k \in E'} \mathbb{P}(e_k, E')$ be the set of conflict edge pairs induced by edges in E'.

A *spanning tree* $T(V_T, E_T)$ of G is a connected subgraph of G such that $V_T = V$, $E_T \subseteq E$ and $|E_T| = n - 1$. The weight of T is denoted by $W(T)$ and it is given by the sum of edges weights in E_T while $\zeta(E_T)$ represents the set of conflict edge pairs present in T and $|\zeta(E_T)|$ the *number of conflicts* in T. When $|\zeta(E_T)| = 0$, we say that T is *conflict free*.

Given the graph G depicted in Figure 1(a), two spanning trees T_1 and T_2 of G are shown in Figure 1(b) and 1(c) with $W(T_1) = 23$ and $W(T_2) = 20$. A *minimum spanning tree* (MST) of G is any spanning tree of G with minimum weight.

The *minimum spanning tree problem with conflict constraints* (MSTC) consists in finding a minimum spanning tree T of G without conflicting edge pairs, that is $\forall\{e_i, e_j\} \in P$ at most one between e_i and e_j belongs to E_T. Obviously MSTC problem is infeasible on the graphs where there are not conflict free spanning trees. For instance, let us consider again graph G in Figure 1(a) and let $P = \{\{(1, 2), (2, 6)\}, \{(1, 2), (5, 6)\}\}$. Since it is not possible to build a conflict free spanning tree of G, with the given set P, MSTC is infeasible on G.

In this paper we face a variant of the MSTC, proposed by Zhang et al. [20], and named Minimum Conflict Weighted Spanning Tree problem (MCWST). In this variant the primary goal consists in finding an MST of G having the minimum number of conflicts and, if this last number is equal to zero, the secondary goal consists in finding a conflict free spanning tree

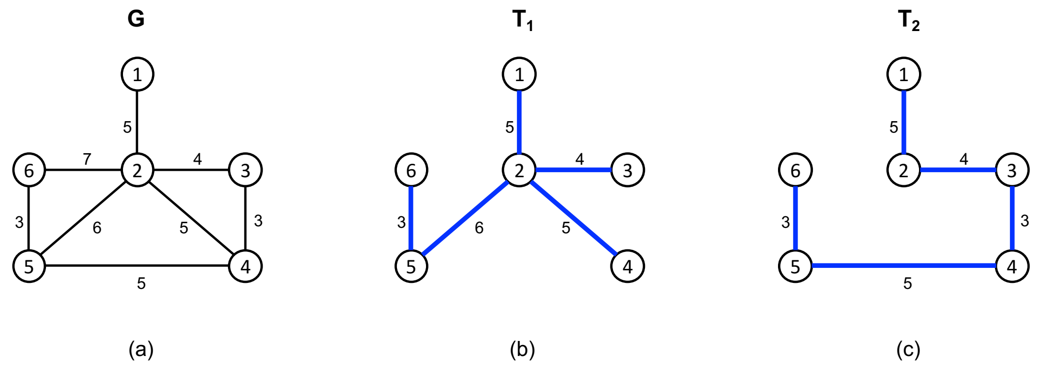

Figure 1: (a) A generic graph G. (b) A spanning tree T_1 of G with $W(T_1) = 23$. (c) A spanning tree T_2 of G with $W(T_2) = 20$

of G with minimum weight. For instance, the trees T_1 and T_2, depicted in Figure 1, are two optimal solutions for the MCWST, with $|\zeta(E_{T_1})| = |\zeta(E_{T_2})| = 1$, when $P = \{\{(1,2),(2,6)\},\{(1,2),(5,6)\}\}$. Since these optimal solutions are not conflict free, their weight is neglected. On the contrary, if $P = \{\{(1,2),(2,6)\}\}$, both T_1 and T_2 are conflict free but T_2 is better than T_1 because $W(T_2) < W(T_1)$ (secondary goal).

From the definitions of MSTC and MCWST, it is easy to see that:

- T^* is an optimal solution of MSTC $\implies$ T^* is an optimal solution for MCWST;
- T^* is an optimal solution of MCWST and $|\zeta(E_{T^*})| = 0$ $\implies$ T^* is an optimal solution for MSTC.

According to the previous observations, by addressing the MCWST problem, we solve even the MSTC problem while, for the instances on which MSTC problem is infeasible, we try to return a spanning tree with the minimum number of conflicts.

3 The Genetic Algorithm

In this section, we introduce our genetic algorithm (GA) we have designed to solve the MCWST. In Section 5, we describe how GA is embedded within a multi ethnic genetic framework to better explore the solution space and to improve its results.

Genetic algorithms, proposed for the first time by J. Holland in 1975 in his book Adaptation in Natural and Artificial Systems [10], are a family of metaheuristics based on the theory of Darwinian natural selection that regulates the biological evolution. While this theory works on a population of individuals, a genetic algorithm operates on a population of feasible solutions, called *chromosomes*, each of them composed of *genes*. The inefficiency of enumerating all feasible solutions recommends fixing in advance the size of solutions space, the *initial population*, on which the algorithm will act. The quality of each chromosome is evaluated by the *fitness function* that corresponds to the objective function or to a function strictly related to the objective function. The aim of the genetic algorithm is to find solutions as close as possible to the optimal one, by combining the chromosomes; this operation is, usually, carried out by crossover operator, which generates new chromosomes (children) by exchanging the genetic material (genes) of their parents. Finally, the mutation operator is applied to the obtained chromosomes. This operator randomly changes one or more genes in each child; it is fundamental to assure the diversity of the children from the parents.

One of the main features of the MSTC is to make it difficult to identify conflict-free trees, whatever is their cost, especially when the number of conflicts in the instance increases. Given this difficulty, we decided to tackle the problem through a genetic algorithm for two main reasons. The first is that this type of algorithm, when appropriately designed, assure a good exploration of the solutions space by increasing our chances to find conflict-free trees. The second reason concerns the cost of the improvement heuristics. Indeed, these heuristics can be time consuming and then they can be applying only on a restricted set of solutions rather than on all the solutions met during the execution of the algorithm. The genetic algorithm provides this set of solutions, that is the final population, containing the solutions survived to the evolutionary process. The results reported in Table 5 show that the application, bounded to the final population only, of these heuristics does not significantly penalize the performance of our algorithm and, on the other side, significantly improves the quality of the final solution.

The main elements of our genetic algorithm are described in the next subsections.

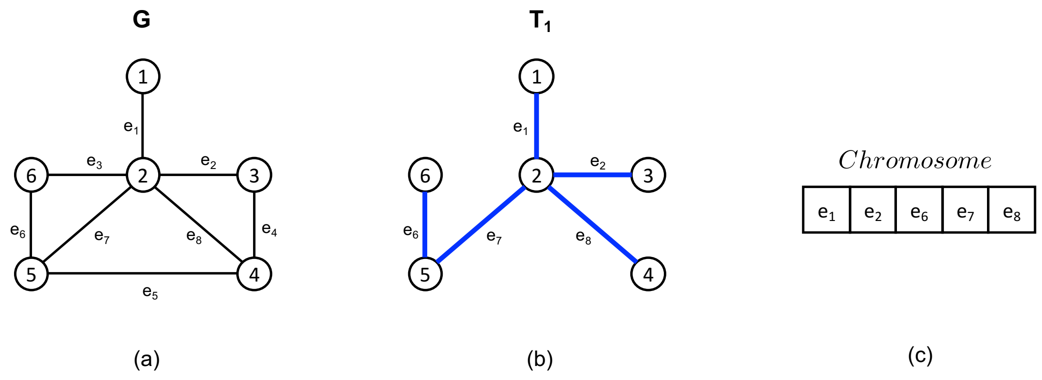

Figure 2: (a) A graph G. (b) A spanning tree T_1 of G. (c) The chromosome representing T_1.

3.1 Chromosome definition and fitness function

The encoding of a feasible solution is the first step of any evolutive algorithm. In GA each chromosome T represents a spanning tree of G and it is encoded by using an array with $|V|-1$ positions (genes) each one containing an edge of E (see Figure 2). In the rest of this paper, we will use the terms spanning tree or chromosome interchangeably.

The quality of each chromosome T is evaluated by a fitness function f. Since the MCWST problem presents two different goals, we designed a fitness function that is able to manage both these goals simultaneously but respecting their priority. More in details, since the primary goal of MCWST is to minimize the number of conflicts, the lower the number of conflicts into the chromosome is, the better its fitness value is. Moreover, given two or more conflict free chromosomes, the secondary goal states that the one with the lowest weight must have a better fitness value. The fitness function that satisfies the previous conditions is defined as follows:

$$f(T) = \begin{cases} |\zeta(E_T)| & \text{if } |\zeta(E_T)| > 0 \\ W(T) - W(T_{max}) & \text{otherwise} \end{cases} \tag{1}$$

where $W(T_{max})$ is an upper bound to the weight of any spanning tree of G. According to Equation (1), the lower the value of the fitness function, the better the quality of the chromosome.

Note that the fitness value of any chromosome with conflicts is greater than zero while the fitness value of any conflict free chromosome is negative because $W(T_{max}) > W(T)$. As a consequence, any conflict free chromosome is always better than any chromosome with at least one conflict (primary goal). Moreover, the lower $W(T)$ is, the lower its fitness value (secondary goal) will be.

For instance, let us consider again the graph G in Figure 1(a) with P={{(1,2),(2,6)}}. An upper bound $W(T_{max})$ can be easily computed by adding the five highest edge weight of G obtaining $W(T_{max}) = 28$. According to the Equation (1), $f(T_1) = W(T_1) - W(T_{max}) = 23 - 28 = -5$ while $f(T_2) = W(T_2) - W(T_{max}) = 20 - 28 = -8$ and therefore T_2 is better than T_1, as expected.

3.2 Initial population

The initial population is composed of *SizePop* different chromosomes randomly generated. More in details, a random weight is assigned to each edge of G and then a minimum spanning tree of G is computed by using Prim's algorithm. If the chromosome obtained is already inside the population then it is rejected because no duplications are allowed. The procedure iterates until either SizePop different chromosomes are found or the threshold *maxD* is reached, where maxD denotes the maximum number of duplicate chromosomes that can be found before stopping the procedure. When this threshold is reached, SizePop is updated to the number of different chromosomes found so far.

The use of the MaxD threshold is necessary because it may happen that either there are no SizePop different spanning trees in G or it is very expensive to identify such trees through a random procedure.

3.3 Creation of new chromosomes

Selection procedure. The selection procedure has the aim to ease the reproduction of individuals with better fitness and, at the same time, the aim to preserve the diversity of the population. In our implementation, the selection of the parents is carried out using a binary tournament strategy. This strategy consists in selecting randomly two chromosomes from the population. The one with the best fitness value is chosen as first parent T_{p1}. The same procedure is used to select the second parent T_{p2} assuring that T_{p2} is different from T_{p1}.

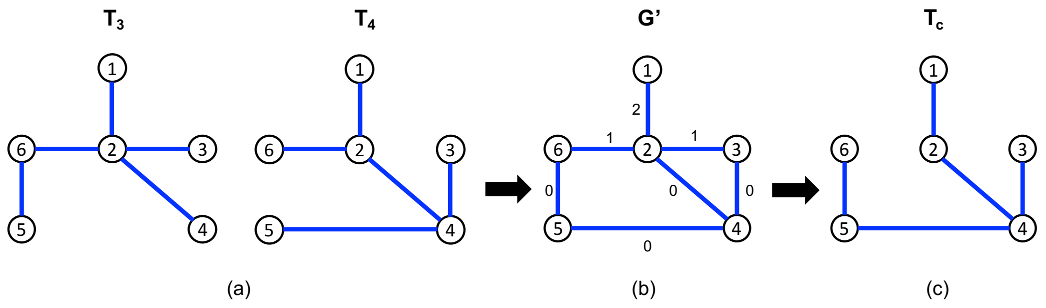

Figure 3: (a) Two spanning trees of G. (b) The subgraph G' generated by crossover. (c) The new child chromosome T_c.

Crossover. The crossover operator is used to generate offspring by recombining the genes of selected parents in order to preserve the characteristics of their genetic heritage. The idea behind our crossover is to build a new spanning tree, by using the edges of parents, in which the number of conflicts is lower than the number of conflicts of parents. To this end, the crossover generates the subgraph $G'(V', E')$ of G induced by edges of both parents, i.e. $E' = E_{T_{p_1}} \cup E_{T_{p_2}}$. Then, the operator associates to each edge $e_i \in E'$ a weight equal to the number of conflicts in which e_i is involved with the other edges in E', i.e. $w(e_i) = |\mathbb{P}(e_i, E')|$. Finally, the crossover computes a minimum spanning tree T_c of G' as new child chromosome.

For instance, let us consider the graph G in Figure 2(a) and its two spanning trees, T_3 and T_4, depicted in Figure 3(a). Moreover, let us suppose that $P = \{\{(1,2),(2,3)\},\{(1,2),(2,6)\}\}$. From T_3 and T_4 the crossover builds the subgraph G' induced by $E_{T_3} \cup E_{T_4}$ (Figure 3(b)) and it associates to each edge $e_i \in E'$ the weight $|\mathbb{P}(e_i, E')|$. Finally, the crossover computes a minimum spanning tree of G' obtaining the tree T_c shown in Figure 3(c).

Mutation. In order to assure that the child chromosome T_c is different from parents, the mutation operator is applied to it. This operator randomly selects one edge in $E \setminus E_{T_c}$ and introduces this edge in T_c generating a cycle. To obtain a new tree, one of the edges in this cycle is randomly selected and removed. In this way, it is assured a differentiation between the child chromosome and the parents, reflecting the natural evolutionary process in which each genetic algorithm is inspired. This operation is carried out a number of times equal to 5% of $|V|$. However, if during the computation a conflict free chromosome is found, the mutation immediately stops and returns this chromosome.

Insertion and Stopping Criteria. If the child chromosome obtained after the mutation operator is already inside the population then it is rejected. Otherwise, the child chromosome will replace one of the $SizePop/2$ worst chromosomes in the population, selected in a random way. As a consequence, the size of the population never changes and the best chromosome found, during the computation, never leaves the population. The genetic algorithm stops when a fixed number ($maxIt$) of iterations is reached.

4 Improvement procedures

In order to improve the best solution found by GA, we use three local search procedures named: *Conflicts Reduction Local Search* (CR), *Weight Reduction Local Search* (WR) and *Neighborhood Weight Reduction Local Search* (NWR). These procedures are invoked on all the chromosomes of the final population. Their aim is either to reduce the conflicts in the chromosomes or to reduce the weight of the conflict free chromosomes. In the following subsections the three procedures are described in details.

4.1 Conflicts Reduction Local Search

The CR procedure is designed to reduce the number of conflicts in the chromosomes and then it is invoked only on the chromosomes with at least one conflict. Given a chromosome T, the first step of the procedure is to identify the edge $e_k \in E_T$ having the maximum number of conflicts with the other edges of E_T, i.e. $e_k = \underset{e_i \in E_T}{\operatorname{argmax}} |\mathbb{P}(e_i, E_T)|$.

The procedure removes e_k from E_T generating a forest composed of two subtrees T_1 and T_2. To obtain a new chromosome T', CR connects T_1 and T_2 by using the edge $e_r \in E \setminus E_T$ where $e_r = \underset{e_i \in E \setminus E_T}{\operatorname{argmin}} |\mathbb{P}(e_i, E_T \setminus \{e_k\})|$. If $|\zeta(E_{T'})| < |\zeta(E_T)|$ the procedure restart from T' otherwise it stops. The new chromosome obtained by CR, if any, replaces T in the population.

4.2 Weight Reduction Local Search

The WR procedure is applied only on the conflict free chromosomes and it tries to minimize their weights without adding conflicts. Given a chromosome T, with $|\zeta(E_T)| = 0$, the procedure starts sorting, in ascending order, the edges in $E \setminus E_T$ according to their weights. Let $\mathbb{L}$ be the list of these sorted edges. At each iteration, the procedure selects from $\mathbb{L}$ the next edge e_k and if $|\mathbb{P}(e_k, E_T)| \leq 1$, it introduces e_k in E_T thus generating a cycle in T. In order to obtain a new chromosome it is necessary to break this cycle by removing one of its edges. There are two cases to consider here:

- $|\mathbb{P}(e_k, E_T)| = 0$
  Let e_j be the edge of the cycle with the maximum weight. Then e_j is removed from $E_T \cup \{e_k\}$ yielding a new conflict free chromosome T'. If $W(T') < W(T)$ then $T \leftarrow T'$, $\mathbb{L} \leftarrow \{e_j\} \cup \mathbb{L} \setminus \{e_k\}$ and WR restarts from the beginning of $\mathbb{L}$. Otherwise the procedure selects the next edge of $\mathbb{L}$.

- $|\mathbb{P}(e_k, E_T)| = 1$
  Let e_j be the edge in conflict with e_k in $E_T \cup \{e_k\}$. If e_j does not belong to the cycle then e_k is removed from the cycle, because no conflicts are allowed in this phase, and the procedure selects the next edge in $\mathbb{L}$. On the contrary, if e_j belongs to the cycle then it is removed yielding a new conflict free chromosome T'. If $W(T') < W(T)$ then $T \leftarrow T'$, $\mathbb{L} \leftarrow \{e_j\} \cup \mathbb{L} \setminus \{e_k\}$ and WR restarts from the beginning of $\mathbb{L}$. Otherwise the procedure selects the next edge of $\mathbb{L}$.

WR stops when no improvements can be obtained or all the edges in $\mathbb{L}$ have been selected.

4.3 Neighborhood Weight Reduction Local Search

The NWR is another procedure used to reduce the weight of conflict free chromosome. Given a conflict free chromosome T, the procedure generates a neighborhood of T as follows. For each edge $e_k \in E \setminus E_T$ such that $|\mathbb{P}(e_k, E_T)| = 0$ or $|\mathbb{P}(e_k, E_T)| = 1$, NWR inserts e_k in E_T yielding a cycle. Now, if $|\mathbb{P}(e_k, E_T)| = 0$ then the procedure breaks the cycle by removing the edge e_j with maximum weight and it produces a new conflict free chromosome T_{e_k}. Otherwise, if $|\mathbb{P}(e_k, E_T)| = 1$, there is an edge $e_j \in E_T$ in conflict with e_k. If e_j belongs to the cycle then NWR removes e_j and it produces a new conflict free chromosome T_{e_k} otherwise no new chromosomes can be obtained in this iteration with the edge e_k. Then e_k is rejected and a new iteration is carried out with the next edge of $E \setminus E_T$. After the selection of all the edges in $E \setminus E_T$, the neighborhood of T is given by: $\mathcal{N}(T) = \bigcup_{e_k \in E \setminus E_T} T_{e_k}$.

After the generation of $\mathcal{N}(T)$, NWR selects the chromosome $T' \in \mathcal{N}(T)$ with the minimum weight. If $W(T') < W(T)$ then $T \leftarrow T'$ and NWR generates the neighborhood of this new chromosome. Otherwise, the procedure stops.

4.4 Local search procedures framework

The three local search procedures described above are applied on the chromosomes of the final population according to the rules shown in the diagram in Figure 4. More in details, let T^* be the best chromosome found by GA and let T be any chromosome of the final population. The following two cases are considered:

- $|\zeta(E_T)| = 0$
  In this case both the procedures, WR and NWR, are invoked on T yielding two new chromosomes T_1 and T_2, respectively. If the lowest fitness between $f(T_1)$ and $f(T_2)$ is better than $f(T^*)$ then T^* is updated accordingly.

- $|\zeta(E_T)| > 0$
  In this case the CR procedure is invoked on T yielding a new chromosome T'. If $|\zeta(E_{T'})| > 0$ but $f(T') < f(T^*)$ then we update T^* with T' and we proceed with the next chromosome into the population. Otherwise, if $|\zeta(E_{T'})| = 0$ the same steps of the previous case are carried out.

5 Multi Ethnic Genetic Approach

The multi ethnic genetic algorithm (Mega) is a technique developed for the genetic algorithms which is aimed at reducing the probability of remaining trapped at a local minimum [4]. The main idea behind the algorithm is to split a starting population in k different sub-populations that, independently,

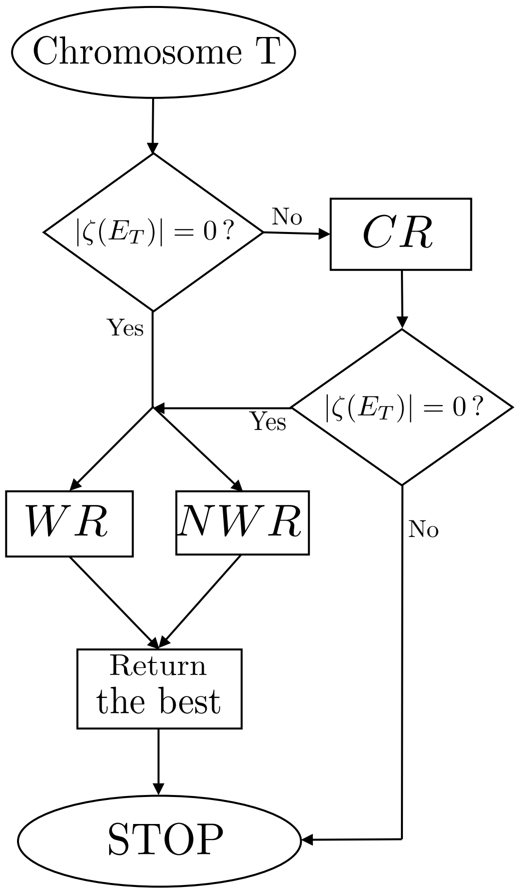

Figure 4: The improvement procedures framework applied on each chromosome of the final population.

evolve in k different environments. The resulting subpopulations are then recombined and the process is iterated, if necessary. Actually, the idea of allowing the simultaneously evolve of k different populations was already adopted in parallel genetic algorithms as the Island Model (see [13, 19]). However, in Mega each population is characterized by its own fitness function that is appropriately selected to diversify the evolution and to carry out a better exploration of the solution space (for more details see [4]).

In the following, we denote by $GA(\cdot)$ the application of our genetic algorithm with the fitness function $\cdot$. Figure 5 displays the framework of Mega.

Mega starts from a single population P_t at time $t = 0$ and it evolves this

$$\begin{tikzcd}
 & \boxed{P_t} \arrow[d, "GA(f)"] & \\
 & \boxed{\hat{P}_t} \arrow[dl, dashed] \arrow[d, dashed] \arrow[dr, dashed] & \\
\boxed{P_{t,1}} \arrow[d, "GA(f_1)"] & \boxed{P_{t,2}} \arrow[d, "GA(f_2)"] & \boxed{P_{t,3}} \arrow[d, "GA(f_3)"] \\
\boxed{\hat{P}_{t,1}} \arrow[dr, dashed] & \boxed{\hat{P}_{t,2}} \arrow[d, dashed] & \boxed{\hat{P}_{t,3}} \arrow[dl, dashed] \\
 & \boxed{P_{t+1}} \arrow[d, "GA(f)"] & \\
 & \boxed{\hat{P}_{t+1}} &
\end{tikzcd}$$

Figure 5: Multi ethnic genetic framework

population by using GA with the fitness function f (equation 1). After this evolutionary step, the obtained evolved population $\hat{P}_t$ is split into $k = 3$ different sub-populations $P_{t,1}$, $P_{t,2}$ and $P_{t,3}$. Differentiation of the environments is induced by slightly changing the fitness function for each sub-population. From f we retrieve three new fitness functions: f_1, f_2 and f_3. Each one of these functions randomly selects the 20% of edges in E_{T^*} and it applies a penalization on them. More in details, f_1 penalizes the selected edges of E_{T^*} by doubling their number of conflicts while f_2 penalizes them by doubling their weight. Finally, f_3 uses the two previous penalizations by doubling both the number of conflicts and the weight of selected edges.

Each of the sub-populations $P_{t,1}$, $P_{t,2}$ and $P_{t,3}$, in turn, evolves according to GA and to a fitness function f_i, with $i = 1, ..., 3$. The three resulting populations $\hat{P}_{t,1}$, $\hat{P}_{t,2}$ and $\hat{P}_{t,3}$ are merged into a unique population P_{t+1}.

Finally, this population is evolved by using GA with f. The best chromosome T^* met during the computation is returned.

After a tuning phase, we set the parameters *SizePop* to 99 and *maxD* to 100. We use a variable value for *maxIt* according to the population on which GA is invoked. In particular, *maxIt* is equal to 2000 when GA is invoked on P_t and 1000 when it is invoked on P_{t+1}. Regarding the sub-populations, *maxIt* is equal to 100 because the size of this population is smaller than the size of P_t and because this value assures us an appropriate diversification of the evolutionary process.

6 Computational Results

In this section we present the computational results of the tests we made in order to evaluate the performance and effectiveness of Mega. The algorithm was coded in C++ on an OSX platform (Imac mid 2011), running on an Intel Core i7-2600 3.4 GHz processor with 8 GB of RAM.

Mega was tested on the benchmark instances presented in [20] and was compared with the tabu search (TS) algorithm proposed in the same paper. These benchmark instances were generated by using a different value for nodes, edges and number of conflicts and they are classified into two types: *type 1* and *type 2*. By construction, there exists at least one conflict free solution for all type 2 instances while type 1 instances may not have conflict free solutions. In order to have a fair comparative study, the CPU time reported in [20] have been scaled according to the Whetstone benchmarks [1].

We now present the results of our tests. The first experiment we carried out is aimed at verifying the stability of Mega by running the algorithm five times on each instance and by comparing the best and average values found. Results are shown in Table 1 that is organized as follows.

The first block of rows is related to instances of Type 1 (instances 1-23) while the second block is related to instances of type 2 (instances 24-50). The first four columns report data on the instances: the identifier (id), the number of nodes (n), the number of edges (m) and the cardinality of P (p). The following two columns, *AvgWeight* and *AvgConf*, show the average weight and the average number of conflicts computed on the five runs while the next two columns report the weight (*Weight*) and the number of conflict (*Conf*) of the best solution found on the five runs. We remind that the best solution is the one with the minimum number of conflicts and, if this solution

	id	n	m	p	AvgWeight	AvgConf	Weight	Conf	Gap
Type 1	1	50	200	199	708	0.0	708	0.0	0.00%
	2	50	200	398	770	0.0	770	0.0	0.00%
	3	50	200	597	917	0.0	917	0.0	0.00%
	4	50	200	995	1365.4	0.0	1336	0.0	0.00%
	5	100	300	448	4099.2	0.0	4088	0.0	0.00%
	6	100	300	897	-	0.6	6095	0.0	100.00%
	7	100	500	1247	4291.2	0.0	4275	0.0	0.00%
	8	100	500	2495	6325	0.0	6199	0.0	0.00%
	9	100	500	3741	7788	0.0	7665	0.0	0.00%
	10	100	300	1344	-	10.4	-	10.0	3.85%
	11	100	500	6237	-	9.4	-	8.0	14.89%
	12	100	500	12474	-	37.8	-	35.0	7.41%
	13	200	600	1797	-	2.2	15029	0.0	100.00%
	14	200	600	3594	-	60.0	-	57.0	5.00%
	15	200	600	5391	-	143.2	-	142.0	0.84%
	16	200	800	3196	22350.8	0.0	22110	0.0	0.00%
	17	200	800	6392	-	27.6	-	23.0	16.67%
	18	200	800	9588	-	88.6	-	87.0	1.81%
	19	200	800	15980	-	177.2	-	172.0	2.93%
	20	300	800	3196	-	55.0	-	52.0	5.45%
	21	300	1000	4995	-	23.4	-	21.0	10.26%
	22	300	1000	9990	-	180.6	-	176.0	2.55%
	23	300	1000	14985	-	330.2	-	329.0	0.36%
Type 2	24	50	200	3903	1636	0.0	1636	0.0	0.00%
	25	50	200	4877	2043	0.0	2043	0.0	0.00%
	26	50	200	5864	2338	0.0	2338	0.0	0.00%
	27	100	300	8609	7434	0.0	7434	0.0	0.00%
	28	100	300	10686	7968	0.0	7968	0.0	0.00%
	29	100	300	12761	8166	0.0	8166	0.0	0.00%
	30	100	500	24740	12652	0.0	12652	0.0	0.00%
	31	100	500	30886	11232	0.0	11232	0.0	0.00%
	32	100	500	36827	11481	0.0	11481	0.0	0.00%
	33	200	400	13660	17728	0.0	17728	0.0	0.00%
	34	200	400	17089	18617	0.0	18617	0.0	0.00%
	35	200	400	20470	19140	0.0	19140	0.0	0.00%
	36	200	600	34504	20716	0.0	20716	0.0	0.00%
	37	200	600	42860	18025	0.0	18025	0.0	0.00%
	38	200	600	50984	20864	0.0	20864	0.0	0.00%
	39	200	800	62625	39895	0.0	39895	0.0	0.00%
	40	200	800	78387	37671	0.0	37671	0.0	0.00%
	41	200	800	93978	38798	0.0	38798	0.0	0.00%
	42	300	600	31000	43721	0.0	43721	0.0	0.00%
	43	300	600	38216	44267	0.0	44267	0.0	0.00%
	44	300	600	45310	43071	0.0	43071	0.0	0.00%
	45	300	800	59600	43125	0.0	43125	0.0	0.00%
	46	300	800	74500	42292	0.0	42292	0.0	0.00%
	47	300	800	89300	44114	0.0	44114	0.0	0.00%
	48	300	1000	96590	71562	0.0	71562	0.0	0.00%
	49	300	1000	120500	76345	0.0	76345	0.0	0.00%
	50	300	1000	144090	78880	0.0	78880	0.0	0.00%

Table 1: Best and average values found by Mega on the type 1 and type 2 instances.

has zero conflicts, with the minimum weight of the tree. Note that, for the solutions with conflicts, we do not report the weight because its value loses meaning in these cases. Finally, the last column *Gap* shows the percentage gap between AvgConf and Conf. This gap is computed with the following formula: $100 \times \frac{AvgConf-Conf}{AvgConf}$.

Let us start the comparison on the instance of type 1. The primary goal is to minimize the number of conflicts. From this point of view, AvgConf and Conf are both equal to zero on the instances 1-9, except 6, and instance 16. Conf is equal to 0 even on the instances 6 and 13 while AvgConf is equal to 0.6 and 2.2 on them. This is an acceptable difference. Unfortunately, the gap values on these last two instances lose meaning because, for any value greater than zero of AvgConf, this gap is always equal to 100%. On the remaining 12 instances, Gap is greater than 5% only 5 times. Regarding the weight values, AvgWeight and Weight have the same value in the first three instances. The maximum gap value is equal to 2.15% and it occurs in the instance 4. In all other cases, the percentage gap is lower than 2%.

On all the instances of type 2, the best and average solutions coincide and these solutions are all conflict free. These results show that type 2 instances are much easier to solve than type 1 instances. A conclusion already reported in [20].

Summarizing, from the results of Table 1 we retrieve that on 30 out of 50 instances the best and the average solutions of Mega coincide in terms of both number of conflicts and weight values. On the remaining 20 instances, the gap on the number of conflicts is only five times greater than 5%, ruling out the “special cases” of instances 6 and 13 where the value is 100% because Conf is equal to zero. Finally, in the few instances where the weight value is different, this gap is very low. According to these results, we can conclude that Mega has a good stability level.

In the next three tables, we compare the results of Mega with the tabu search TS proposed in [20]. Following [20], Table 2 contains the subset of type 1 instances on which the TS found a conflict free solution. Unfortunately, for instance 6, only the CPU time but not the weight is reported.

The first four columns show the characteristics of the instance as already mentioned for Table 1. The column Opt/Best reports the optimal solution value found by mathematical model described in [20] and implemented in CPLEX. A time limit of 5000 seconds is fixed and whenever CPLEX reaches this limit, the solution value is reported with the symbol “*”. This means that CPLEX did not certify the optimality of the solution and the value

ID	n	m	p	Opt/Best	TS		Mega		GapOpt	GapTS
					Weight	Time	Weight	Time		
1	50	200	199	708	711	1.17	708	0.71	0.00%	0.42%
2	50	200	398	770	785	1.13	770	0.68	0.00%	1.91%
3	50	200	597	917	1086	0.98	917	0.63	0.00%	15.56%
4	50	200	995	1324	1629	1.16	1336	0.66	0.91%	17.99%
5	100	300	448	4041	4207	6.33	4088	2.39	1.16%	2.83%
6	100	300	897	6523*	-	5.99	6095	1.81	6.56%	-
7	100	500	1247	4275	4539	17.71	4275	5.18	0.00%	5.82%
8	100	500	2495	6653*	6812	17.09	6199	5.12	6.82%	9.00%
9	100	500	3741	-	8787	14.94	7665	3.72	-	12.77%

Table 2: Type 1 Problems-Feasible.

reported is an upper bound of the optimal solution. Moreover, if CPLEX does not find a feasible solution, within the time limit, the symbol "-" is shown. The next four columns report the weight (*Weight*) of the tree and the CPU time (*Time*), in seconds, of TS and Mega, respectively. Since both the algorithms always find conflict free solutions on this set of instances, we do not report a column with the number of conflicts. The last two columns show the percentage gap between the weight of Mega and the optimal solution (*GapOpt*) and between the weight of Mega and TS (*GapTS*). These percentage gaps are computed with the formulas: $100 \times \frac{Opt - Mega(Weight)}{Opt}$ and $100 \times \frac{TS(Weight) - Mega(Weight)}{TS(Weight)}$, respectively.

By comparing the solutions of Mega with the optimal ones, we can see that our algorithm optimally solves the instances 1, 2, 3 and 7 while GapOpt is equal to 0.91% and 1.16%, on the instances 4 and 5, respectively. On the instances 6 and 8, the solution values of Mega are significantly lower than the upper bounds found by CPLEX. Finally, on the instance 9, Mega finds a feasible solution in 4 seconds while CPLEX did not in 5000 seconds. These results prove the effectiveness of Mega because, often, it finds the optimal solution or a solution very close to the optimal one.

The results of Table 2 show that Mega overcomes TS in terms of computational time and solution quality. Indeed, the solutions found by Mega are always better than the solutions found by TS with a percentage gap that ranges from 0.42% (instance 1) to 17.99% (instance 4). In particular, on 5 out of 8 instances GapTS is greater than 5.8%. Finally, TS never finds an optimal solution while Mega does it four times. The computational time spent on these instances is low for both algorithms but Mega is always faster

id	**n**	**m**	**p**	**TS**		**Mega**		**Gap**
				Conf	**Time**	**Conf**	**Time**	
10	100	300	1344	13	6.77	10	2.69	23.08%
11	100	500	6237	11	15.17	8	4.98	27.27%
12	100	500	12474	41	14.64	35	6.68	14.63%
13	200	600	1797	2	72.48	0	12.23	100.00%
14	200	600	3594	67	70.24	57	21.71	14.93%
15	200	600	5391	149	80.12	142	29.43	4.70%
16	200	800	3196	2	105.21	0	23.42	100.00%
17	200	800	6392	39	98.01	23	28.20	41.03%
18	200	800	9588	95	97.10	87	35.32	8.42%
19	200	800	15980	178	104.93	172	44.48	3.37%
20	300	800	3196	63	239.63	52	62.68	17.46%
21	300	1000	4995	38	303.04	21	83.68	44.74%
22	300	1000	9990	207	345.25	176	117.58	14.98%
23	300	1000	14985	351	381.28	329	134.42	6.27%
Avg				**89.7**	**138.1**	**79.4**	**43.4**	

Table 3: Type 1 Problems-Feasible Unknown.

than TS.

Table 3 shows the results of GA and TS on the remaining type 1 instances on which TS never finds conflict free solutions. The first four columns show the characteristics of the instance. The next four columns report the number of conflicts (*Conf*) and the CPU time (*Time*) of TS and Mega, respectively. The column *Gap* shows the percentage gap between the Conf values of two algorithms. This percentage gap is computed with the formula $100 \times \frac{TS(Conf)-Mega(Conf)}{TS(Conf)}$.

We remind that for the type 1 instances it is not guaranteed the presence of a conflict free spanning tree. However, Mega certified the presence of a conflict free solution on two of these instances (13 and 16) while TS finds solutions with two conflicts on these instances. Behind these two cases, it is evident that Mega is more effective than TS because it always finds better solutions. Ruling out the instances 13 and 16, on the remaining 12 instances the Gap value ranges from 3.37% to the 44.74% and this gap is 8 times greater than 14%.

Regarding the performance, Mega is always faster than TS. More in details, Mega solves all the instances up to 200 nodes in less than a minute

	id	n	m	p	Opt	TS	Mega	Gap
Type 2	24	50	200	3903	1636	1.32	0.46	65.15%
	25	50	200	4877	2043	1.93	0.47	75.69%
	26	50	200	5864	2338	1.56	0.51	67.36%
	27	100	300	8609	7434	8.03	1.88	76.58%
	28	100	300	10686	7968	7.47	1.68	77.51%
	29	100	300	12761	8166	7.88	1.72	78.18%
	30	100	500	24740	12652	19.29	3.30	82.90%
	31	100	500	30886	11232	16.77	3.48	79.24%
	32	100	500	36827	11481	15.16	3.51	76.85%
	33	200	400	13660	17728	30.27	7.25	76.05%
	34	200	400	17089	18617	38.63	7.49	80.61%
	35	200	400	20470	19140	26.64	7.40	72.22%
	36	200	600	34504	20716	82.49	11.17	86.46%
	37	200	600	42860	18025	96.16	11.35	88.20%
	38	200	600	50984	20864	122.67	12.38	89.91%
	39	200	800	62625	39895	117.33	16.35	86.07%
	40	200	800	78387	37671	106.52	15.70	85.26%
	41	200	800	93978	38798	105.42	16.02	84.80%
	42	300	600	31000	43721	112.03	18.61	83.39%
	43	300	600	38216	44267	153.88	21.28	86.17%
	44	300	600	45310	43071	98.99	24.32	75.43%
	45	300	800	59600	43125	214.50	31.86	85.15%
	46	300	800	74500	42292	191.63	34.31	82.10%
	47	300	800	89300	44114	245.12	34.25	86.03%
	48	300	1000	96590	71562	301.27	39.81	86.79%
	49	300	1000	120500	76345	287.49	31.60	89.01%
	50	300	1000	144090	78880	325.16	36.11	88.89%
Avg						**101.32**	**14.60**	

Table 4: CPU Time of TS and Mega on the Type 2 instances.

while, in the same time, TS solves only the first three instances. The maximum time spent by Mega is 135 seconds on the instance 23 while TS requires 381 seconds on the same instance. According to the average values reported on the last line of the table, we can state that Mega is three times faster than TS and the solutions provided by Mega are 11.5% better than the solutions of TS.

The last comparison is carried out on the type 2 instances and the results

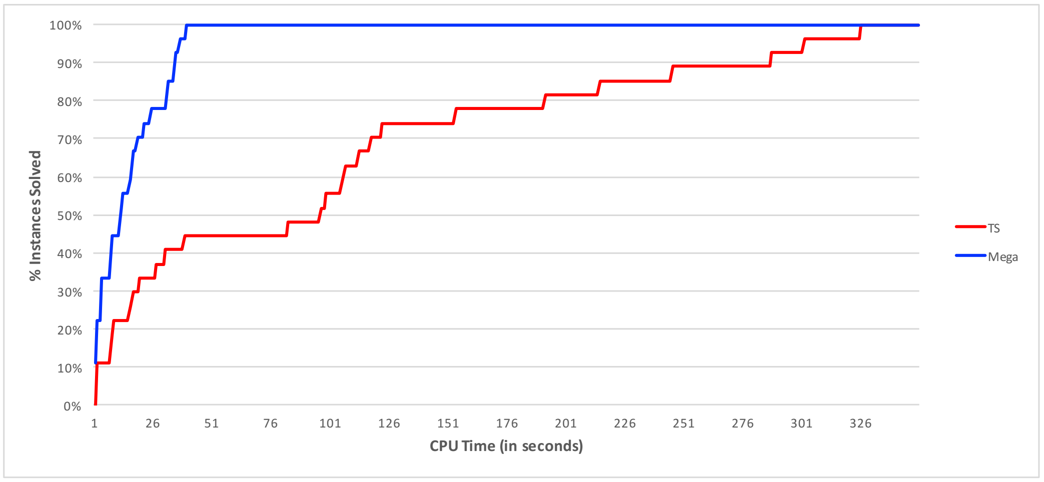

Figure 6: Performance comparison of TS vs Mega from the results of Table 4.

are shown in Table 4. On these instances both the algorithms always find the optimal solution. All these optimal solutions are conflict free and their weight is reported into the column Opt. For this reason, we compare only the CPU times in this table. The first four columns are the same as the previous table. The next three columns report the optimal weight (Opt) and the CPU time of TS and Mega, respectively. Finally, the column *Gap* shows the percentage gap values of the CPU times.

It is evident from the values of column Mega that type 2 instances are easier to solve than type 1 instances because all these instances are optimally solved in less than 37 seconds. Once again, Mega results are always generated faster than these generated by TS with a percentage gap that is always greater than 65%. In particular, this gap grows as the size of instance grows. Indeed, on the instances with 200 or more nodes Gap is almost always greater than 80%. The average values reported on the last line of the table show the relevant difference between the two algorithms from the performance point of view.

In Figure 6 we represent the results of Table 4 in a way that better highlights the performance of two algorithms. The horizontal axis represents the CPU time in seconds and the vertical axis represents the percentage of instances solved within a fixed CPU time. This means that the faster the curve grows, the better the performance of the algorithm is. The blue curve is associated with Mega and the red curve with TS. It is evident from this figure that the blue curve grows much faster than the red curve. Indeed, Mega reaches the 100% of instances solved in around 36 seconds. In the same time (36 seconds) TS solves around 45% of the instances while it requires 326 seconds to reach the 100% of solved instances.

Finally, in order to highlight the impact of improvement procedures on the performance and effectiveness of Mega, we implemented a version of Mega without these procedures and we named it NoImpr. Table 5 contains the results of the comparison between Mega and NoImpr. The first block of rows is related to instances of Type 1 (instances 1-23) while the second block is related to instances of type 2 (instances 24-50). The first four columns report data on the instances. The following six columns report the average weight (*AvgWeight*), the average number of conflicts (*AvgConf*) and the average computational time (*AvgTime*) of Mega and NoImpr, respectively. Regarding the effectiveness of these two algorithms, it is obvious that the solutions of Mega are always better than or equal to the solution of NoImpr. However, if we focus the attention on the instances where both the algorithms find conflict free solutions (id 1-5,7,8,24-32,40,44), we discover that the percentage gap of the solution values is lower than 2%. This means that the improvement procedures do not produce a significant improvement on these instances. The real contribution carried out by improvement procedures concerns the capacity of significantly reducing the number of conflicts present in a solution. Indeed, Mega finds a solution conflict free 36 times while NoImpr finds only 16 conflict free solutions. In particular, for all the 27 type 2 instances NoImpr provides a conflict free solution while NoImpr does that only 11 times. By comparing the values of AvgConf columns it is easy to see that effectiveness of the improvement procedures regarding the reduction of the conflicts. In particular, on the instances where both the algorithms do not find a conflict free solution (id 6,12-17,19-25), the percentage gap between the number of conflicts ranges from 6.88% (23) to the 70% (6 and 13). The only drawback derived from the application of improvement procedures is the increment of the computational time. It is evident from the column AvgTime that NoImpr is significantly faster than Mega˙Indeed, in the worst case, NoImpr requires 21 seconds (21) while Mega requires 141 seconds (23). However, since the time requires by Mega is always lower than two and half minutes we are satisfied by its performance and, in particular, by the quality of the solutions that this additional time provides us with respect to NoImpr.

					MEGA			NoImpr		
	id	**n**	**m**	**p**	**AvgWeight**	**AvgConf**	**AvgTime**	**AvgWeight**	**AvgConf**	**AvgTime**
Type 1	1	50	200	199	708	0.0	0.7	708	0.0	0.5
	2	50	200	398	770	0.0	0.7	770	0.0	0.5
	3	50	200	597	917	0.0	0.6	917	0.0	0.5
	4	50	200	995	1365.4	0.0	0.6	1388	0.0	0.5
	5	100	300	448	4099.2	0.0	2.4	4116.8	0.0	1.5
	6	100	300	897	-	0.6	1.9	-	2.0	1.3
	7	100	500	1247	4291.2	0.0	5.3	4329.8	0.0	2.6
	8	100	500	2495	6325	0.0	5.0	6427.2	0.0	2.2
	9	100	500	3741	7788	0.0	3.2	-	2.2	2.0
	10	100	300	1344	-	10.4	2.9	-	13.2	1.2
	11	100	500	6237	-	9.4	5.0	-	15.2	1.8
	12	100	500	12474	-	37.8	6.9	-	42.2	1.6
	13	200	600	1797	-	2.2	12.6	-	7.4	7.3
	14	200	600	3594	-	60.0	24.0	-	73.2	5.5
	15	200	600	5391	-	143.2	31.6	-	157.6	4.7
	16	200	800	3196	22350.8	0.0	22.2	-	0.4	9.9
	17	200	800	6392	-	27.6	28.4	-	38.4	7.5
	18	200	800	9588	-	88.6	38.3	-	104.6	6.0
	19	200	800	15980	-	177.2	48.0	-	194.4	5.3
	20	300	800	3196	-	55.0	62.8	-	66.8	16.8
	21	300	1000	4995	-	23.4	79.3	-	39.2	21.0
	22	300	1000	9990	-	180.6	119.1	-	200.6	15.8
	23	300	1000	14985	-	330.2	141.3	-	354.6	13.8
Type 2	24	50	200	3903	1636	0.0	0.5	1636	0.0	0.4
	25	50	200	4877	2043	0.0	0.5	2043	0.0	0.4
	26	50	200	5864	2338	0.0	0.6	2338	0.0	0.4
	27	100	300	8609	7434	0.0	1.9	7479	0.0	1.2
	28	100	300	10686	7968	0.0	1.8	8016.4	0.0	1.1
	29	100	300	12761	8166	0.0	1.9	8181.4	0.0	1.1
	30	100	500	24740	12652	0.0	3.4	12793.2	0.0	1.8
	31	100	500	30886	11232	0.0	3.6	11281.2	0.0	1.9
	32	100	500	36827	11481	0.0	3.6	11580.6	0.0	1.9
	33	200	400	13660	17728	0.0	7.7	-	0.8	4.1
	34	200	400	17089	18617	0.0	7.7	-	0.4	4.2
	35	200	400	20470	19140	0.0	7.6	-	2.0	4.3
	36	200	600	34504	20716	0.0	11.6	-	2.6	5.5
	37	200	600	42860	18025	0.0	12.0	-	13.2	5.8
	38	200	600	50984	20864	0.0	12.5	-	5.4	6.1
	39	200	800	62625	39895	0.0	16.7	-	2.4	6.9
	40	200	800	78387	37671	0.0	15.9	38307.8	0.0	7.1
	41	200	800	93978	38798	0.0	17.3	-	4.0	7.6
	42	300	600	31000	43721	0.0	19.2	-	13.0	12.5
	43	300	600	38216	44267	0.0	22.3	-	0.8	13.2
	44	300	600	45310	43071	0.0	24.6	43434.4	0.0	14.2
	45	300	800	59600	43125	0.0	32.7	-	2.4	15.7
	46	300	800	74500	42292	0.0	35.6	-	15.6	16.4
	47	300	800	89300	44114	0.0	35.4	-	30.2	17.1
	48	300	1000	96590	71562	0.0	41.5	-	8.4	17.9
	49	300	1000	120500	76345	0.0	33.4	-	16.4	18.9
	50	300	1000	144090	78880	0.0	36.9	-	17.6	20.8

Table 5: Comparison between Mega and NoImpr.

7 Conclusions

In this paper, we studied the minimum conflict weighted spanning tree problem, and we developed a genetic algorithm to solve it and three local search procedures to improve the quality of the solution found. To obtain a better exploration of the solution space, we embedded the genetic algorithm in a multi ethnic genetic framework.

The computational results show that Mega often finds the optimal solution or a solution close to the optimal one. Moreover, it found two new conflict free solutions with respect to the best known solutions in the literature. Finally, Mega significantly outperforms the tabu search heuristic, proposed in the literature, both in terms of computational time and quality of the solutions found.

References

[1] Whetstone benchmarks.

[2] F. Carrabs, R. Cerulli, M. Gaudioso, and M. Gentili. Lower and upper bounds for the spanning tree with minimum branch vertices. *Computational Optimization and Applications*, 56(2):405–438, 2013.

[3] F. Carrabs, R. Cerulli, R. Pentangelo, and A. Raiconi. Minimum spanning tree with conflicting edge pairs: a branch-and-cut approach. *Annals of Operations Research. To appear.*, 2018.

[4] C. Cerrone, R. Cerulli, and M. Gaudioso. Omega one multi ethnic genetic approach. *Optimization Letters*, 10(2):309–324, 2016.

[5] C. Cerrone, R. Cerulli, and A. Raiconi. Relations, models and a memetic approach for three degree-dependent spanning tree problems. *European Journal of Operational Research*, 232(3):442–453, 2014.

[6] R. Cerulli, M. Gentili, and A. Iossa. Bounded-degree spanning tree problems: Models and new algorithms. *Computational Optimization and Applications*, 42(3):353–370, 2009.

[7] A. Darmann, U. Pferschy, and J. Schauer. Determining a minimum spanning tree with disjunctive constraints. *Lecture Notes in Computer*

Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5783 LNAI:414–423, 2009.

[8] A. Darmann, U. Pferschy, J. Schauer, and G.J. Woeginger. Paths, trees and matchings under disjunctive constraints. *Discrete Applied Mathematics*, 159(16):1726–1735, 2011.

[9] C. Feremans, M. Labbé, and G. Laporte. The generalized minimum spanning tree problem: Polyhedral analysis and branch-and-cut algorithm. *Networks*, 43(2):71–86, 2004.

[10] J. H. Holland. Adaption in natural and artificial systems. *Ann Arbor MI: The University of Michigan Press*, 222(3):2–5, 1975.

[11] M. M. Kanté, C. Laforest, and B. Momège. Trees in graphs with conflict edges or forbidden transitions. *Theory and Applications of Models of Computation: 10th International Conference, TAMC 2013, Hong Kong, China, May 20-22, 2013. Proceedings*, pages 343–354, 2013.

[12] A. Klein, D. Haugland, J. Bauer, and M. Mommer. An integer programming model for branching cable layouts in offshore wind farms. *Advances in Intelligent Systems and Computing*, 359:27–36, 2015.

[13] B.D.H Latter. The island model of population differentiation: a general solution. *Genetics*, 73(1):147–157, 1973.

[14] T. Oncan, R. Zhang, and A.P. Punnen. The minimum cost perfect matching problem with conflict pair constraints. *Computers and Operations Research*, 40(4):920–930, 2013.

[15] U. Pferschy and J. Schauer. The knapsack problem with conflict graphs. *Journal of Graph Algorithms and Applications*, 13(2):233–249, 2009.

[16] U. Pferschy and J. Schauer. The maximum flow problem with disjunctive constraints. *Journal of Combinatorial Optimization*, 26(1):109–119, 2013.

[17] R. Sadykov and F. Vanderbeck. Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing*, 25(2):244–255, 2013.

[18] P. Samer and S. Urrutia. A branch and cut algorithm for minimum spanning trees under conflict constraints. *Optimization Letters*, 9(1):41–55, 2014.

[19] D. Whitley, S. Rana, and R.B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology*, 7(1):33–48, 1999.

[20] R. Zhang, S.N. Kabadi, and A.P. Punnen. The minimum spanning tree problem with conflict constraints and its variations. *Discrete Optimization*, 8(2):191–205, 2011.