Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Local Search Algorithms for the Minimum Cardinality Dominating Trail Set of a Graph

Paolo Detti, Carlo Meloni, Marco Pranzo

RT-DIA-84-2003 **October 2003**

detti@dii.unisi.it, meloni@deemail.poliba.it, mpranzo@dia.uniroma3.it

ABSTRACT

Given a graph $G = (V, E)$, a *dominating trail* D_T in G is a trail such that each edge of G has at least one endpoint belonging to it (i.e., a dominating trail *covers* all the edges of G). A *dominating trail set* is a collection of edge-disjoint trails that altogether cover all the edges of G. A *minimum dominating trail set* ($MDTS$) is a dominating trail set of minimum cardinality. The problem of finding a $MDTS$ in a graph G is related to the hamiltonian completion number of the line graph of G, i.e., the minimum number of edges to be added to its line graph $L(G)$ to make it Hamiltonian. The $MDTS$ problem is known to be NP–hard for general graphs, whereas efficient algorithms exist for particular classes of graph. In this paper, a local search approach is proposed. Extensive computational experiments are carried on a wide set of instances. The results allow us to point out the behavior of the proposed algorithms with respect to both the quality of the solutions and the required execution time.

1 Introduction

Given a graph $G = (V, E)$, a *trail* is a sequence $T := (v_0, e_0, v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k)$, where $(v_0, v_1, v_2, \ldots, v_k)$ are nodes of G, $(e_0, e_1, e_2, \ldots, e_{k-1})$ are distinct edges of G, and v_i and v_{i+1} are the endpoints of e_i for $0 \leq i \leq k-1$. The trail is a *path* if its nodes $(v_0, v_1, v_2, \ldots, v_k)$ are distinct. In other words, a trail is a path that can pass more times through the same node. A path or a trail may consist of a single node.
A *dominating trail* D_T in G is a trail such that each edge of G has at least one endpoint belonging to it (i.e., a dominating trail *covers* all the edges of G). Note that a dominating trail may not exist on G. A *dominating trail set* Σ is a collection of edge-disjoint trails that altogether cover all the edges of G. A *minimum dominating trail set* ($MDTS$) is a dominating trail set of minimum cardinality.
The problem of finding a $MDTS$ is closely related to the Hamiltonicity of a graph. A graph G is called Hamiltonian if it has a Hamiltonian path. The problem of finding the minimum number of edges which need to be added to G to make it Hamiltonian is known in literature as the problem of finding the *Hamiltonian completion number* of a graph and it is usually denoted as $HCN(G)$. In particular, $MDTS$ is related to $HCN(G)$ restricted to a particular class of graphs, called *line graphs*. The line graph $L(G)$ of $G = (V, E)$ is a graph having $|E|$ nodes, each node of $L(G)$ being associated to an edge of G. There is an edge between two nodes of $L(G)$ if the corresponding edges of G are adjacent. Linear-time algorithms exist for recognizing a line graph $L(G)$ and obtain its *root* graph G [20, 28]. Harary and Nash-Williams [16] link the problem of finding $HCN(L(G))$ and $MDTS$ showing that the line graph $L(G)$ of a graph G has a Hamiltonian path if and only if G has a dominating trail. As a consequence, if $HCN(L(G)) = k$ then the cardinality of $MDTS$ of G is $k + 1$.

The paper is organized as follows. In Section 2 the related literature is discussed and some applications are reported. Section 3 presents some constructive heuristics, while in Section 4 a metaheuristic approach for the problem of finding a $MDTS$ on a graph G is proposed. Computational experiments are made on a wide set of instances as reported in Section 5. The results allow us to point out the behavior of the proposed algorithms with respect to both the quality of the solutions and the execution time. Finally, some conclusions follows in Section 6.

2 Literature review and applications

The problem of finding $HCN(G)$ on a line graph is well known to be NP–hard [6]. Particular special conditions on G have been found that ensure the existence of a Hamiltonian path on $L(G)$ [34], and therefore $HCN(L(G)) = 0$. When G is a tree or a forest the problem may be solved in linear time [14, 19, 25, 29, 30], while an approximate algorithm for the weighted version of the problem was proposed by Wu et al. [37].
When G is an interval graph [26], a circular-arc graph [7], a block graph [32, 36, 38], a bipartite permutation graph [32] or a cograph [21], it was shown that there exist polynomial time algorithms for finding $HCN(G)$.
Raychaudhuri [27] presented a $O(|V|^5)$ algorithm for finding $HCN(G)$ when G is the line graph of a tree, while Agnetis et al. [3] proposed a linear algorithm for this case.

For Cactus graphs, i.e., graphs in which every edge is part of at most one cycle in G, Detti and Meloni [10] proposed a linear time algorithm for finding $MDTS$ or $HCN(L(G))$.
Agnetis et al. [2] showed the NP–hardness of the problem of finding $HCN(L(G))$ even when G is bipartite, and proposed, for this case, a heuristic approach.
The study of line graphs is strongly related to important graph invariants, i.e., the *interval number*, the *total interval number* and the *Wiener index* [15, 18, 27].
Finding a $MDTS$ of a graph (or HCN of a line graph) is often required in routing and sequencing problems [2, 1, 9, 24], and in graph searching and in data structures updating [11, 12, 13].
Some applications in production and manufacturing systems exist for the problem of finding $MDTS$. In [2], a problem arising in a real industrial context is addressed, namely the coordination between two consecutive production stages. A graph formulation has been presented for the problem, and a solution approach, based on finding a $MDTS$ on a bipartite graph, is proposed. In [1], a production system in which two competitive users share a common resource to execute a set of processes is studied. The problem of sequencing the processes on the resource is formulated and efficiently solved by finding a $MDTS$ on *augmented trees*.

3 Constructive heuristics for $MDTS$

In this section we briefly describe some constructive heuristic approaches for $MDTS$. The first heuristic, proposed by Agnetis et al. [2], is called DOMWALKS, the second heuristic is based on an exact polynomial algorithm for $MDTS$ on trees [3]. The third approach consists of simple greedy procedures based on minimal knowledge of the graph.

3.1 DOMWALKS heuristic

Agnetis et al. [2] proposed a heuristic (called DOMWALKS) with the aim to build a dominating trail set Σ, trying to keep its cardinality as low as possible. The approach is based on the following considerations. If graph G happens to have an eulerian trail, the solution to $MDTS$ is straightforward, since an eulerian trail is a dominating trail. If an eulerian trail does not exist, the idea is to remove from G some edges so that, in the resulting graph $\bar{G}$, the degree of each node is even. If $\bar{G}$ is connected, it has an eulerian cycle, and hence an eulerian trail. An eulerian trail T_e in $\bar{G}$ is a dominating trail in G, since each removed edge is adjacent to some edge of T_e. If $\bar{G}$ is not connected, each component either has an eulerian cycle or consists of a single node. All the edges of each component can be dominated by an eulerian trail, but a new trail may need when moving from one component to another.

One key feature of this approach is the edge removal phase. It is performed by finding an *odd-vertex pairing* (OVP) of minimal length. Given the graph G, an OVP Π is a set of edge-disjoint (but possibly not node-disjoint) paths having nodes with odd degree in G as endpoints, such that each node with odd degree in G is the endpoint of exactly one path, and no nodes of even degree are endpoints of any path. When the algorithm removes Π from G, a graph $\bar{G}$ is obtained in which all vertices have even degree. Hence, if an OVP with few edges is found, chances are that $\bar{G}$ is still connected. Therefore, the algorithm seeks an OVP with a minimum total number of edges. Such a pairing can be found in

polynomial time by solving a matching problem [2].

In the first step of the algorithm (Figure 1) an OVP Π is found. $\bar{G}$ is obtained from G by removing the edges of Π. The algorithm then starts from an empty trail set, and iteratively adds trails one by one until all the edges of G are dominated. The current set of trails formed by the algorithm is denoted by Σ_C. The algorithm builds each trail by finding a dominating trail for one or more connected components of $\bar{G}$. In order to limit the number of used trails as much as possible, the algorithm tries to use the edges of the OVP Π to move from the current component to another one. Of course, no edge of Π can be used more than once throughout the algorithm, so it may be the case that no path of Π is anymore available to move from the current component to another. In this case a trail is completed and added to the current trail set Σ_C, the algorithm selects a not visited component, and a new trail is started.

Algorithm DOMWALKS
Compute the odd-vertex pairing (OVP) Π of minimum total length;
Let $\bar{G} := G - \Pi$;
if $\bar{G}$ is eulerian **then** Σ:={eulerian trail on $\bar{G}$}, STOP.
else
begin

$T = \emptyset$;
while a not visited eulerian component $\hat{G}$ of $\bar{G}$ exists **do**
begin

if a not visited component $\hat{G}' \neq \hat{G}$ such that a path $(u, \ldots, v)$, $u \in \hat{G}$, $v \in \hat{G}'$ of the OVP exists
then

append to T a trail dominating $\hat{G}$, $\hat{G}'$ and $(u, \ldots, v)$, set $\hat{G} = \hat{G}'$;

else

append to T a trail dominating $\hat{G}'$, $\Sigma = \Sigma \cup T$, $T = \emptyset$;

end
while a not dominated edge e exists in a path $p \in \Pi$ **do**

let p' be the not dominated subpath of p; $\Sigma = \Sigma \cup p$;

while a not visited and not dominated component $\hat{G}$ of $\bar{G}$ exists **do**

let T be a trail dominating $\hat{G}$; $\Sigma = \Sigma \cup T$;

end

Figure 1: Algorithmic scheme of DOMWALKS

3.2 DOMTREE-based heuristic

An efficient algorithm ($O(|V|)$), based on the theoretical relationship between $MDTS$ and the Hamiltonian Completion Number problem (HCN), has been proposed for $MDTS$ on trees [3]. It is known as DOMTREE.
In the DOMTREE-based heuristic for general graphs G DOMTREE is applied to a spanning tree ST of G. Since the edges not belonging to ST possibly may be not dominated

Maximum Degree Greedy and Minimum Degree Greedy Algorithms
begin
  while $E \neq \emptyset$ **do**
    Let v_i be the node with maximum (minimum) degree $d(v_i)$.
    Let $e_i = (v_i, v_x)$ be an arc incident to v_i.
    Add to the trail set Σ the trail $T = \{v_i, e_i, v_x\}$.
    Remove all the edges incident to v_i.
  end
end

Figure 2: Algorithmic scheme of MaxDG (MinDG) algorithm.

by the trails dominating ST, these edges are "linked" to some trail in order to give a low cardinality trail set which dominates the overall graph G. As spanning trees can be efficiently individuated, the overall procedure for $MDTS$ based on DOMTREE results as a fast algorithm.

3.3 Greedy algorithms

We have developed two simple greedy algorithms that can be used as initial solution for a local search based metaheuristic. The Maximum Degree Greedy algorithm (MaxDG) and the Minimum Degree Greedy algorithm (MinDG) produce a trail set Σ. They work as shown in Figure 2. At each step the algorithm identifies the vertex v_i with maximum (minimum) degree $d(v_i)$, and a new trail T is generated. The new trail is composed of a single edge adjacent to the selected vertex v_i, and it is added to Σ. In these algorithms only the edges incident to node v_i are considered dominated by the new trail T. The dominated edges are "removed" from the graph and the node degrees are updated correspondently. This process is repeated until all the edges of the graph are dominated.

From the quality point of view the solution generated by the MaxDG and MinDG procedures may have a poor quality, since a large number of trails are generated. However such solutions can be easily improved by a local search procedure.

4 A metahuristic approach to the $MDTS$ problem

In this section we describe a metaheuristic approach developed to tackle the $MDTS$ problem. A metaheuristic is an iterative solution procedure, which combines subordinate heuristic tools into a more sophisticated framework. A master process guides and modifies the operations of these heuristics in order to efficiently produce high-quality solutions. It may manipulate a complete or incomplete single solution, or a collection of solutions at each iteration. Also the subordinate heuristics can be constructive, enumerative or iterative procedures or hybrids. Such methods cover a large family of techniques, varying from rather simple to very sophisticated (see, e.g., [35], [8]).
The *Iterated Local Search* (ILS) is a conceptually very simple metaheuristic and it is able to obtain state-of-the-art performance for several hard combinatorial problems. We mention here the Iterated Lin-Kernighan [17] [23] and the Chained Lin-Kerighan [4] for

the TSP, the paper of Balas and Vazacopoulos [5] for job shop scheduling, the ILS [33] for permutation flow shop scheduling, and the recent work of Smyth et al. [31] for the MAX-SAT. All these algorithms fall in the ILS general framework. For a tutorial on the Iterated Local Search procedures see [22].

The section is organized as follows. First, some notations are introduced; then, we present two different neighborhood functions, namely $\mathcal{N}_1$ and $\mathcal{N}_2$ for the $MDTS$ problem; next, a simple local search procedure based on the proposed neighborhoods is described; finally, the Iterated Local Search metaheuristic is introduced.

4.1 Notations

It is useful to introduce some notations about the structure of the dominating trail set Σ. A dominating trail set Σ has the property to partition the set of edges of G. In fact, each edge $e \in E$ either belongs to a trail of Σ or it is dominated by (at least) one of them. Without loss of generality, we assume each edge as dominated exactly by one trail of Σ, i.e., in the case an edge is dominated by more than one trail we consider only one of them. In view of this assumption we associate to each trail $T \in \Sigma$ a sequence $S(T)$ of edges (i.e., $S(T)$ contains T and edges dominated by T) in which subsequent edges have always one node in common. Note that a trail T can be associated to different edge sequences and a dominating trail set Σ can produce different families of edge-partitioning sequences. Each sequence S has two *ending nodes* and two *ending edges*, referred as *left* and *right* nodes and edges, respectively. Let v_r and e_r denote the right ending node and the right ending edge, respectively. The ending edge $e_r = (v_r, v_x)$ is adjacent to an edge in the sequence $S(T)$ sharing node v_x with it. We define the *right ending subsequence* of $S(T)$ as the sequence of edges $E_r(T)$ starting by e_r and having the property of sharing node v_x. Observe that each ending subsequence $E_r(T)$ contains a *right terminal subsequence* (possibly reduced to a single edge) $\sigma_r(T)$ that can be permuted holding the property that subsequent edges must have one node in common. The same definitions can be considered for the left end of the sequence $S(T)$ leading to the definition of the *left ending subsequence* $E_l(T)$, and of the *left terminal subsequence* $\sigma_l(T)$. Therefore each $T \in \Sigma$ has associated a sequence S and a *terminal subsequence* $\sigma(T) = \sigma_r(T) \cup \sigma_l(T)$ containing edges that can allow to link up a sequence associated to another trail in Σ.

4.2 Neighborhood $\mathcal{N}_1$ and $\mathcal{N}_2$

A neighborhood structure $\mathcal{N}(\Sigma)$ specifies for each solution Σ which solutions are "close" in some sense. The solutions $\Sigma' \in \mathcal{N}(\Sigma)$ are called neighbors of Σ and they can be reached from Σ. Given a solution Σ, we consider the neighborhood $\mathcal{N}_1(\Sigma)$ obtained by merging two trails having two adjacent edges in their terminal subsequences, i.e., sharing a common node. More formally, given two trails T_1 and T_2, let $e_h \in \sigma(T_1)$ and $e_k \in \sigma(T_2)$ be the two terminal edges sharing the node v_i. The two terminal subsequences $\sigma(T_1)$ and $\sigma(T_2)$ can be permuted in such a way that the correspondent ending edges become e_k and e_h respectively. Hence, the trails T_1 and T_2 can be merged in a single trail leading to a decrease of the objective function $|\Sigma|$.

In Figure 3 (a), a graph G is represented showing a dominating trail set Σ, containing four trails (the bold edges). Moreover for each trail T_i the set $\sigma(T_i)$ is indicated. Note that there exists a node shared by two terminal subsequences, namely $\sigma(T_2)$ and $\sigma(T_4)$.

Hence a move in $\mathcal{N}_1$ is possible. In Figure 3 (b) the same graph is shown after a move has been done in the neighborhood $\mathcal{N}_1$. In fact the trails T_2 and T_4 are now merged together.

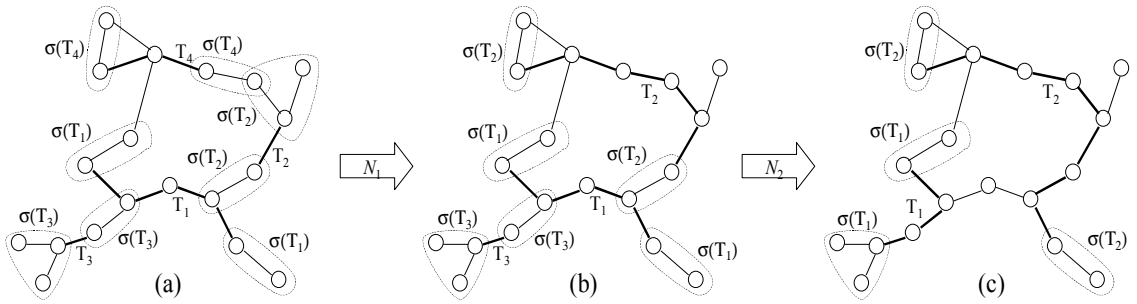

Figure 3: A move in $\mathcal{N}_1$ and $\mathcal{N}_2$.

Given a solution Σ the neighborhood $\mathcal{N}_2(\Sigma)$ is obtained by splitting a trail into two trails and merging them with two different pre-existent trails. More precisely, let $\Sigma = \{T_0, T_1, \ldots, T_q\}$ and let us suppose that two consecutive edges e_i and e_{i+1} in $S(T_0)$ share a node with an element of $\sigma(T_1)$ and $\sigma(T_2)$, respectively. Then it is possible to split trail T_0 into two trails and merge them with trails T_1 and T_2. Clearly, after applying a move in the neighborhood $\mathcal{N}_2$, the objective function $|\Sigma|$ decreases by one.

In Figure 3 (c), an example of a move in $\mathcal{N}_2$ applied to the graph in Figure 3 (b) is shown. In this case, trail T_1 is split and merged with T_3 and T_2.

4.3 A Local Search Algorithm

Based on the definition of the neighborhood $\mathcal{N}_1$ and $\mathcal{N}_2$ we define a Local Search (LS) algorithm, see Figure 4. The Local Search performs a first-improvement neighborhood exploration in the neighborhood $\mathcal{N}_1$ until a local minimum is reached. At this point a move is performed using a first-improvement exploration of the neighborhood $\mathcal{N}_2$ and a new local search in $\mathcal{N}_1$ is executed. The Local Search procedure stops when no moves both in $\mathcal{N}_1$ and $\mathcal{N}_2$ are possible. Since the Local Search accepts only improving moves this algorithm converges towards a local minimum in a small number of iterations.

4.4 The Iterated Local Search algorithm

It is possible to improve the performances of the Local Search algorithm introduced in the previous subsection by applying an Iterated Local Search framework. The basic scheme of an ILS is given in Figure 5. The main idea behind the ILS is to perturbate the incumbent solution and to apply a new local search starting from the perturbated solution. A new local minimum is obtained and then the Evaluation Criterion decides whether to accept the new solution as a new incumbent solution or to discard it. The Iterated Local Search consists of three different phases, the Perturbation, the Local Search and the Acceptance Criterion. The Perturbation phase applies a perturbation to the incumbent solution obtaining a new candidate solution. This phase is very important in constructing an effective ILS algorithm. In the Local Search phase the perturbed solution is improved by means of a local search procedure. It has to be pointed out that every local search algorithm, even a very complex one, could be used in the local search phase. Finally the Acceptance Criterion decides either to accept the new candidate solution, and the search

Local Search (LS)
Given an initial solution
begin
  while a move is executed **do**
    search for a move in $\mathcal{N}_1$
    if a move is found **then** apply the move
    else
      begin
        search for a move in $\mathcal{N}_2$
        if a move is found **then** move in $\mathcal{N}_2$
      end
  end
end

Figure 4: Algorithmic scheme of LS.

Iterated Local Search (ILS)
Given an initial solution
begin
  Local Search
  while Stopping Criterion **do**
    Perturbation
    Local Search
    Acceptance Criterion
  end
end

Figure 5: Algorithmic scheme of ILS.

process continues from the new solution, or to discard it. In the latter case the search process continues from the last accepted solution. In the remaining of this Section we describe in details all the parts of the proposed ILS algorithm.

- As *Initial Solution* we consider the solution obtained by some constructive algorithm, in our tests we considered as initial solution the solutions obtained by MinDG, MaxDG, DOMTREE heuristics.

- In the *Local Search phase*, we use as local search the LS algorithm, introduced in Section 4.3.

- The *Perturbation phase* applies random moves to the incumbent solution. A perturbation move splits randomly a certain number of trails. Hence in a perturbed solution there are more terminal subsequences σ, and then LS has greater chances to improve the solution quality. A perturbation is characterized by a parameter $p \in (0, 1)$ called *perturbation strength* which indicates the maximum number of random moves that can be applied to the incumbent solution. The actual number of

perturbating moves is randomly chosen in $[1, p * |E|]$.

- As *Acceptance Criterion* we consider a *Better* acceptance, i.e., a solution is accepted as the new incumbent solution only if it strictly decreases $|\Sigma|$.

- The *Stopping Criterion* of the ILS algorithm is bounded by a maximum number of iterations. Moreover the ILS algorithm could stop even earlier if for the incumbent solution $|\Sigma| = LB$ holds, i.e., the algorithm certifies the incumbent solution as an optimal solution. The Lower Bound LB used to stop the search process is inspired to theoretical results reported by Agnetis et al. [3]. In their work they use the concept of *marked nodes*. Marking a node means that at least one trail of Σ must pass through that node. Let v_{ln} be a leaf node of G, i.e., $d(v_{ln}) = 1$, if the adjacent node v_m of v_{ln} has exactly degree $d(v_m) = 2$ then, in order to dominate the edge (v_{ln}, v_m), a trail passing in v_m is required to exist in every feasible dominating trail set Σ. That is node v_m has to be marked. We refer to it as a *marked leaf*. Since every trail has two endvertices the number N_{ml} of marked leaves gives a simple way to calculate a lower bound for the cardinality of the $MDTS$, i.e., $LB = max\{1, \lceil N_{ml}/2 \rceil\}$. In Figure 6 a graph with $LB = 2$ is illustrated. The use of the lower bound leads to much shorter computational requirements in many instances. In other words, the search process is stopped whenever the optimality of the solution is proved or when a maximum number of Perturbation - Local Search iterations is performed.

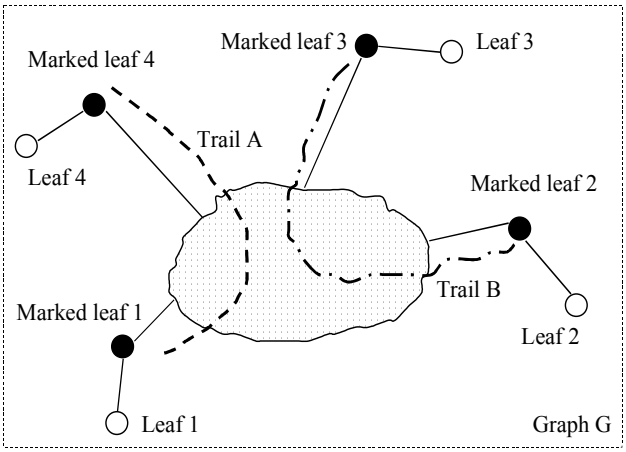

Figure 6: A graph G with $N_{ml} = 4$ and $LB = 2$.

5 Computational experiments

In this section we describe the experiments carried out to evaluate the performance of the proposed algorithms. In particular, two local search algorithms (LS) and (ILS) are compared with the DOMWALKS heuristic [2], which is able to produce very good quality solutions in strict computational times.

In a preliminary test phase we tuned the two proposed algorithms. We tested the influence of the initial solution, of the perturbation strength p. The best configurations obtained for the two algorithms are:

- LS, as defined in Section 4.3, starting from a solution obtained by the MinDG algorithm.
- ILS, as introduced in Section 4.4, with a maximum perturbation strength $p = 0.15$, 10000 iterations as stopping criterion, the Better acceptance criterion and MinDG as initial solution.

Quite surprisingly the local search procedures are able to achieve the best results using MinDG as initial solution, even if MaxDG usually performs better than MinDG as stand-alone heuristic. The reason of this finding seems to be that MinDG gives better building blocks allowing the Local Search procedure to reach easily good local optima.

In order to ensure a fair comparison between the LS, ILS and DOMWALKS all the experiments have been run on the same machine: a 350 MHz Pentium II processor equipped with 128 MB ram. Moreover all the algorithm are coded in C.

5.1 Instance description

The algorithms have been tested on three sets of randomly generated problems. The first two sets are the randomly generated bipartite graphs introduced in [2], while the third set contains general random graphs.

The first set of instances, denoted as Set 1, consists of balanced bipartite graphs $G = (V_1, V_2, E)$, i.e., graphs in which the sets V_1 and V_2 have the same cardinality. The instances in this set are generated, varying the cardinality $n = |V_1| = |V_2|$ from 10 to 100 and the graph density from 5% to 40%. The values 10, 30, 60, 80, 100 have been considered for n, and 5, 10, 20, 30, 40% for the density d. For each pair (n, d) 20 connected instances have been generated.

The second set of instances (Set 2) consists of randomly generated bipartite graphs in which a dominating trail *does* exist. In these instances $N = |V_1| + |V_2|$ varies from 10 to 100 and 90 instances for each value of N are generated. Note that these graphs are not balanced, that is in general $|V_1| \neq |V_2|$.

In the third set (Set 3), general graphs $G = (V, E)$ have been generated. Also in this case, as done for Set 1, $|V|$ varies from 10 to 100, and the density d varies from 5% to 40%. Moreover, for each pair $(|V|, d)$ 20 connected instances are collected.

5.2 Performance analysis

In Tables 1–11, the performance of DOMWALKS, LS and ILS on the three test sets are reported. In each table, columns 2–4, 5–7 and 8–10 describe the behavior of DOMWALKS, LS and ILS, respectively. In particular, the cardinality of the trail set found ($|\Sigma|$), the computational time in seconds (time) and the percentage of proven optimal solutions (%opt) are reported, for each algorithm. Note that each row refers to the average over 20 instances for Set 1 and Set 3, whereas the average is over 90 instances for Set 2.
Since the perturbation phase of the ILS algorithm is random performed, the results of the ILS are average values related to five runs of the algorithm.

	DOMWALKS			LS			ILS		
d	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.1	3.4	0.14	15%	2.45	<0.01	5%	1.43	0.16	57%
0.2	2.1	0.14	40%	2.10	<0.01	20%	1.17	0.07	83%
0.3	1.5	0.14	70%	1.90	<0.01	30%	1.05	0.02	95%
0.4	1.2	0.14	80%	1.35	<0.01	65%	1	<0.01	100%

Table 1: Set 1: $n = 10$ instances.

In Tables 1–5, the results on Set 1 are presented. In column 1, the density value d is reported. In each row, the entries reported in columns 2–7 are average values on the 20 instances, while the entries reported in columns 8–10 are average values related to the 20 instances and the 5 runs of ILS. On this set, ILS outperforms DOMWALKS and LS in terms of solution quality, whereas LS is the faster algorithm. DOMWALKS is able to attain solutions of better quality with respect to LS. The computational times of DOMWALKS are almost independent from the graph density d, i.e., from the number of edges, and they increase almost linearly with the number of nodes. On the other hand, the computational times of LS increase both with the graph density d and the number of nodes $|V|$. However, LS results the fastest algorithm with an average computational time of about 0.15 seconds. From computational point of view, LS is able to find an optimal solution for dense graphs, whereas it fails for sparse instances ($d \leq 0.1$). Since ILS employs LS, the computational times of ILS for dense graphs are almost the same of LS. When tackling sparse instances the computational requirements of ILS increase up to few seconds; only in these cases DOMWALKS performs faster than ILS. Nevertheless, ILS is able to prove the optimality of the solution in almost all the instances (95.6%).
It has to be pointed out that on small and sparse instances (Table 1) all algorithms are not able to certify the optimality of the solutions. That is due to the poor quality of the lower bound in these cases.

In Table 6, the results on the bipartite instances with a dominating trail are shown (Set 2). In column 1, N is the total number of nodes. The entries reported in each row are average values on the 90 instances (and on the five 5 runs for ILS). ILS is able to solve all the instances requiring an average computational time of 0.03 seconds. DOMWALKS solve almost all the instances, and its computational times increase slowly as the instance dimension grows.

Finally, in Tables 7–11 the performances on the general graphs are reported (Set 3). Also in this case, ILS produces solutions with best quality and LS results the fastest algorithm. With respect to the bipartite instances of Set 1 these instances seem to be easier to solve. In fact, all the algorithms require less computational times and are able to yield better solutions.

In Table 12, the average computational times and the average percentage of proven optimal solution for each test set are reported.

6 Conclusions

In this paper the problem of finding a $MDTS$ on a graph is addressed. A local search approach is proposed, compared with algorithms from literature and tested on a wide set of instances. The results show the effectiveness of the proposed approach with respect

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.1	1.9	0.20	40%	2.75	<0.01	0%	1.02	0.22	98%
0.2	1	0.20	100%	1.25	<0.01	75%	1	<0.01	100%
0.3	1	0.20	100%	1	<0.01	100%	1	<0.01	100%
0.4	1	0.20	100%	1	<0.01	100%	1	<0.01	100%

Table 2: Set 1: $n = 30$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	2.35	0.48	25%	3.65	0.02	0%	1.25	4.33	75%
0.1	1.05	0.48	95%	1.45	0.03	60%	1	0.52	100%
0.2	1	0.47	100%	1	0.05	100%	1	0.05	100%
0.3	1	0.47	100%	1	0.08	100%	1	0.08	100%
0.4	1	0.48	100%	1	0.10	100%	1	0.10	100%

Table 3: Set 1: $n = 60$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	1.7	0.81	55%	3.20	0.05	0%	1.07	7.30	93%
0.1	1	0.81	100%	1.35	0.08	65%	1	2.46	100%
0.2	1	0.81	100%	1	0.17	100%	1	0.17	100%
0.3	1	0.82	100%	1	0.27	100%	1	0.28	100%
0.4	1	0.83	100%	1	0.39	100%	1	0.39	100%

Table 4: Set 1: $n = 80$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	1.2	1.31	80%	2.35	0.12	0%	1	8.94	100%
0.1	1	1.33	100%	1.05	0.20	95%	1	0.88	100%
0.2	1	1.32	100%	1	0.44	100%	1	0.45	100%
0.3	1	1.32	100%	1	0.70	100%	1	0.71	100%
0.4	1	1.35	100%	1	0.94	100%	1	0.94	100%

Table 5: Set 1: $n = 100$ instances.

N	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
10	1.01	0.22	97.7%	1.753	<0.01	34%	1	<0.01	100%
20	1.10	0.20	92.2%	2.000	<0.01	40%	1	<0.01	100%
30	1.44	0.18	97.7%	2.322	<0.01	41%	1	<0.01	100%
40	1.22	0.24	97.7%	2.600	<0.01	46%	1	<0.01	100%
50	1	0.34	100%	2.800	<0.01	51%	1	0.01	100%
60	1	0.28	100%	3.011	<0.01	55%	1	0.02	100%
70	1	0.42	100%	3.322	0.01	61%	1	0.03	100%
80	1	0.35	100%	3.522	0.01	61%	1.002	0.06	99.7%
90	1	0.45	100%	3.744	0.02	66%	1	0.07	100%
100	1	0.49	100%	4.178	0.03	63%	1	0.17	100%

Table 6: Set 2: Results on instances with dominating trail.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.1	2.25	0.13	30%	1.75	<0.01	35%	1.04	<0.01	96%
0.2	1.8	0.14	45%	1.70	<0.01	35%	1	<0.01	100%
0.3	1.45	0.13	70%	1.60	<0.01	40%	1	<0.01	100%
0.4	1.1	0.13	90%	1.35	<0.01	70%	1	<0.01	100%

Table 7: Set 3: $|V| = 10$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.1	1.6	0.15	60%	2.30	<0.01	20%	1	<0.01	100%
0.2	1	0.15	100%	1.40	<0.01	70%	1	<0.01	100%
0.3	1	0.15	100%	1	<0.01	100%	1	<0.01	100%
0.4	1	0.15	100%	1	<0.01	100%	1	<0.01	100%

Table 8: Set 3: $|V| = 30$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	1.95	0.21	45%	2.95	<0.01	5%	1.20	0.03	97%
0.1	1	0.21	100%	1.40	<0.01	60%	1	0.02	100%
0.2	1	0.20	100%	1	0.01	100%	1	0.01	100%
0.3	1	0.21	100%	1	0.01	100%	1	0.01	100%
0.4	1	0.20	100%	1	0.01	100%	1	0.01	100%

Table 9: Set 3: $|V| = 60$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	1.95	0.21	45%	2.45	<0.01	5%	1	0.42	100%
0.1	1	0.21	100%	1	0.01	100%	1	0.01	100%
0.2	1	0.27	100%	1	0.02	100%	1	0.02	100%
0.3	1	0.27	100%	1	0.03	100%	1	0.04	100%
0.4	1	0.28	100%	1	0.05	100%	1	0.05	100%

Table 10: Set 3: $|V| = 80$ instances.

d	DOMWALKS			LS			ILS		
	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt	$\|\Sigma\|$	time	%opt
0.05	1.1	0.35	90%	1.70	0.01	60%	1	0.35	100%
0.1	1	0.35	100%	1.05	0.03	95%	1	0.04	100%
0.2	1	0.36	100%	1	0.05	100%	1	0.06	100%
0.3	1	0.37	100%	1	0.08	100%	1	0.09	100%
0.4	1	0.37	100%	1	0.13	100%	1	0.13	100%

Table 11: Set 3: $|V| = 100$ instances.

	DOMWALKS		LS		ILS	
	time	%opt	time	%opt	time	%opt
Set 1	0.62	82.6%	0.15	65.8%	1.22	95.6%
Set 2	0.31	98.5%	<0.01	51.8%	0.03	99.9%
Set 3	0.22	85.8%	0.01	73.6%	0.05	99.8%
Total	0.38	88.9%	0.05	63.7%	0.43	98.4%

Table 12: Summary of results.

to both the quality of the solutions and the execution times. In particular, over the 1820 considered instances, ILS is able to prove the optimality of the solutions in more than 98% on average. Future research includes the study of better lower bounds for the problem, the design of exact algorithmic approaches for general graph and the study of exact polynomial algorithms for special classes of graphs.

References

[1] Agnetis, A., Detti, P., Meloni, C., (2003), *Process selection and sequencing in a two-agents production system*, 4OR, 1 (2), 103–119.

[2] Agnetis, A., Detti, P., Meloni, C., Pacciarelli D., (2001), *Set-up coordination between two stages of a supply chain*, Annals of Operations Research, 107, 15–32,

[3] Agnetis, A., Detti, P., Meloni, C., Pacciarelli D., (2001), *A linear algorithm for the Hamiltonian completion number of the line graph of a tree*, Information Processing Letters, 79, 17–24.

[4] Applegate, D., Cook, W., Rohe, A., (2003), *Chained Lin-Kernighan for Large Traveling Salesman Problems*, INFORMS Journal on Computing, 15 (1), 82–92.

[5] Balas, E., Vazacopoulos, A., (1998), *Guided local search with shifting bottleneck for job shop scheduling*, Management Science, 44 (2), 262–275.

[6] Bertossi, A.A., (1981), *The edge Hamiltonian problem is NP-hard*, Information Processing Letters, 13, 157–159.

[7] Bonuccelli, M.A., Bovet, D.P., (1979), *Minimum node disjoint path covering for circular-arc graphs*, Information Processing Letters, 8(4), 159–161.

[8] Corne, D., Dorigo, M., Glover, F. (Eds.), (1999), *New Ideas in Optimization*, McGraw-Hill, London.

[9] Detti, P., Meloni, C., (2001), *Part type selection and batch sequencing in a two-stage manufacturing system*, Proceedings of 16th International Conference on Production Research, Praha.

[10] Detti, P., Meloni, C., (2003), *A linear algorithm for the Hamiltonian completion number of the line graph of a cactus*, Discrete Applied Mathematics, to appear.

[11] De Vitis, A., (1997), *The cactus representation of all minimum cuts in weighted graph*, Technical Report, 454, IASI-CNR, Roma.

[12] Fleischer, L., (1999), *Building chain and cactus representations of all minimum cuts from Hao–Orlin in the same asymptotic run time*, Journal of Algorithms, 33 (1), 51–72.

[13] Fomin, F.V., Golovach, P.A., (2000), *Graph searching and interval completion*, SIAM Journal of Discrete Mathematics, 13 (4), 454–464.

[14] Goodman, S.E., Hedetniemi, S.T., Slater, P.J., (1975), *Advances on the Hamiltonian Completion Problem*, Journal of the ACM, 22 (3), 352–360.

[15] Gutman, I., (1998), *Buckley-type relations for Wiener-type structure-descriptors*, Journal of the Serbian Chemical Society, 63 (7), 491-496.

[16] Harary, F., Nash-Williams, C.St.J.A., (1965), *On Eulerian and Hamiltonian graphs and line-graphs*, Canadian Mathematics Bulletin, 8, 701–709.

[17] Johnson, D.S., McGeoch, L.A., (1997) *The travelling salesman problem: a case study in local optimization*, Local Search in Combinatorial Optimization, John Wiley and Sons, New York, 215–310.

[18] Klavzar, S., Gutman, I., (1997), *Wiener number of vertex-weighted graphs and a chemical application*, Discrete Applied Mathematics, 80 (1), 73–81.

[19] Kundu, S., (1976), *A linear algorithm for the Hamiltonian completion number of a tree*, Information Processing Letters, 5, 55–57.

[20] Lehot, P.G.H., (1974), *An optimal algorithm to detect a line graph and output its root graph*, Journal of the ACM, 21, 569-575.

[21] Lin, R., Olariu, S., Pruesse, G., (1995), *An Optimal Path Cover Algorithm for Cographs*, Computers and Mathematics with Applications, 30 (8), 75–83.

[22] Lourenço, H.R.D., Martin, O., Stützle, T., (2002), *Iterated Local Search* In Glover, F., Kochenberger, G. (Eds.), Handbook of Metaheuristics, Kluwer, 321–353.

[23] Martin, O., Otto, S.W., (1996), *Combining simulated annealing with local search heuristics*, Annals of Operations Research, 63, 57–75.

[24] Meloni, C., (2001), *An evolutionary algorithm for the sequence coordination in furniture production*, Lecture Notes in Computer Science, 2264, 91-105.

[25] Misra, J., Tarjan, R.E., (1975), *Optimal chain partitions of trees*, Information Processing Letters, 4 (1), 24–26.

[26] Rao Arikati, S., Pandu Rangan, C., (1990), *Linear algorithm for optimal path cover problem on interval graphs*, Information Processing Letters, 35, 149–153.

[27] Raychaudhuri, A., (1995), *The total interval number of a tree and the Hamiltonian completion number of its line graph*, Information Processing Letters, 56, 299–306.

[28] Roussopoulos, N.D., (1973), *A max {m,n} algorithm for determining the graph H from its line graph G*, Information Processing Letters, 2, 108-112.

[29] Skupień, Z., (1975), *Path Partitions of Vertices and Hamiltonity of Graphs*, in: Fiedler, M. (Ed.), Recent Advances in Graph Theory (Proceedings of the 2nd Czechoslovakian Symposium on Graph Theory, Prague 1974), Akademia, Praha, 481–491.

[30] Skupień, Z., (1976), *Hamiltonian Shortage, path partitions of vertices, and matchings in a graph*, Colloquium Mathematicum, 36 (2), 305–318.

[31] Smyth, K., Hoos, H.H., Stützle, T., (2003), *Iterated Robust Tabu Search for MAX-SAT*, Proceedings of the Sixteenth Canadian Conference on Artificial Intelligence (AI'2003), 2003 (to appear).

[32] Srikant, R., Sundaram, R., Singh, K.S., Pandu Rangan, C., (1993), *Optimal path cover problem on block graphs and bipartite permutation graphs*, Theoretical Computer Science, 115, 351–357.

[33] Stützle, T., (1998), *Applying iterated local search to the permutation flow shop problem*, Technical Report AIDA-98-04, FG Intellektik, TU Darmstadt.

[34] Veldman, H.J., (1988), *A Result on Hamiltonian Line Graphs Involving Restrictions on Induced Subgraphs*, Journal of Graph Theory, (12) 3, 413–420.

[35] Voß, S., Martello, S., Osman, I.H., Roucairol, C. (Eds.), (1999), *Meta-Heuristics:Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers.

[36] Wong, P.K., (1999), *Optimal path cover problem on block graphs*, Theoretical Computer Science, 225, 163–169.

[37] Wu, Q.S., Lu, C.L., Lee, R.C.T., (2000), *An Approximate Algorithm for the Weighted Hamiltonian Path Completion Problem on a Tree*, Lecture Notes in Computer Science, 1969, 156–167, Springer Berlin.

[38] Yan, J.H., Chang, G.J., (1994), *The path-partition problem in block graphs*, Information Processing Letters, 52, 317–322.