J Heuristics (2013) 19:797–818
DOI 10.1007/s10732-013-9224-z

Applying local search to the feedback vertex set problem

Philippe Galinier · Eunice Lemamou · Mohamed Wassim Bouzidi

Received: 21 November 2011 / Revised: 26 March 2013 / Accepted: 27 May 2013 /
Published online: 11 June 2013

Abstract The feedback vertex set problem (FVSP) consists in making a given directed graph acyclic by removing as few vertices as possible. In spite of the importance of this NP-hard problem, no local search approach had been proposed so far for tackling it. Building on a property of acyclic graphs, we suggest in this paper a new representation of the solutions of the FVSP (feedback sets). Thanks to this solution representation, we are able to design a local transformation (equivalent to a neighborhood) that changes a feedback set into a new one. Based on this neighborhood, we have developed a simulated annealing algorithm for the FVSP. Our experiments show that our algorithm outperforms the best existing heuristic, namely the greedy adaptive search procedure by Pardalos et al.

1 Introduction

In a directed graph, a *feedback set* is a set of vertices that intersects any cycle of the graph. Given a directed graph, the goal of the *Feedback Vertex Set Problem* (FVSP) is to find a feedback set with a minimum cardinality. In other words, the problem consists in making the graph acyclic by removing as few vertices as possible. The FVSP has applications in several areas, including program verification (Seymour 1995), deadlock resolution, and Bayesian inference (Yehuda et al. 1994).

The FVSP is NP-hard (Garey and Johnson 1979; Yannakakis 1978). This problem has been extensively studied (see Festa et al. 2009 for a recent survey), especially from the standpoint of approximation algorithms (Bafna et al. 1994; Becker and Geiger 1979; Monien and Schultz 1981; Qian et al. 1996). On the other hand, very few

P. Galinier (✉) · E. Lemamou · M. W. Bouzidi
Ecole Polytechnique de Montréal,
Montreal, QC, Canada
e-mail: philippe.galinier@polymtl.ca

heuristics have been proposed in order to tackle the FVSP. To the best of our knowledge, the only heuristics described in the literature are greedy heuristics and the Greedy Randomized Adaptive Search Procedure (*GRASP*), both proposed in Pardalos et al. (1999) and Festa et al. (2001).

So far, local search has never been applied to the FVSP. Applying local search to a combinatorial optimization problem (such as the FVSP) necessitates to define a search space (the set of configurations), an evaluation function and a neighborhood function—such a triplet is what we name a *local search approach* in the following. Assuming that a configuration is simply a feedback set, and the evaluation function (to be minimized) the cardinality of the set, it is still necessary to define an appropriate local operation (a move mechanism) that transforms a feedback set into a new one. If we consider an operation that simply transforms a feedback set into a new set of vertices (for example, simply inserting or removing a vertex, or swapping two vertices), it is costly in general to determine whether the new set is a feedback set or not—as it is already noticed in Pardalos et al. (1999). The above considerations illustrate the difficulties encountered if one wants to design a neighborhood for the FVSP; this explains why no local search approach was known so far for tackling this problem.

In this paper, a practical local search approach is proposed for the first time for tackling the FVSP. This approach relies on a new representation of the configurations. While configurations still correspond to legal feedback sets, they are no longer represented as sets. Thanks to the proposed solution representation, it becomes possible to design a practical move mechanism. Based on this local search approach, we have developed a simulated annealing algorithm, named the Simulated Annealing for the Feedback Vertex Set Problem, *SA-FVSP* in short. Experiments conducted with *SA-FVSP* show that our algorithm performs much better than *GRASP*.

The rest of the paper is organized as follows. A review of literature is first presented in Sect. 2. The proposed local search approach is described in Sect. 3. The *SA-FVSP* algorithm is detailed in Sect. 4. Section 5 is devoted to computational results. Finally, concluding remarks are given in Sect. 6.

2 Related work

Feedback set problems include several interrelated problems whose goal is to find a minimum-weight (or minimum cardinality) set of vertices (or arcs) that intersect all the cycles of a given graph. There are different versions of the problem, depending on whether the graph is directed or undirected, and the vertices (arcs) are weighted or unweighted. These problems have been extensively studied in the literature—the recent survey by Festa et al. (2009) devoted to them quotes about a hundred citations. In particular, active research has been conducted from the standpoint of approximation algorithms (Bafna et al. 1994; Becker and Geiger 1979; Monien and Schultz 1981; Qian et al. 1996). The particular problem tackled in this paper is the feedback vertex set problem (FVSP), whose goal is to find a minimum-cardinality set of vertices that intersects any cycle in a given (unweighted) directed graph. This problem is (in its decision version) one of the 21 problems proven to be NP-complete by Karp in 1972 (Garey and Johnson 1979; Yannakakis 1978). Several exact algorithms have been proposed in the literature, notably a branch and bound algorithm developed in Lin and Jou (1999). On the other hand, very few heuristics have been designed for this problem. The only heuristics described so far in the literature are greedy heuristics and the Greedy Randomized Adaptive Search Procedure (*GRASP*), both proposed in Pardalos et al. (1999) and Festa et al. (2001).

Graph reduction plays an important role in solving FVSP because it improves the efficiency of both exact algorithms and heuristics. A contraction (reduction) operation reduces the original graph while it preserves the information necessary for finding the minimum feedback set. Five contraction operations have been proposed in Levy and Low (1988). More recently, three new contraction operations have been presented in Lin and Jou (1999). Reduction has been exploited in exact algorithms, notably Lin and Jou (1999), and in heuristics (Pardalos et al. 1999). In the following of this section, we detail the description of the *GRASP* heuristic because we will use the results of *GRASP* as a basis of comparison for those of our own heuristic.

The principle of the *GRASP* metaheuristic (Feo and Resende 1995) is to generate a solution with a randomized greedy heuristic, and then to improve its quality thanks to a refinement procedure (typically a local search heuristic). This process is reiterated a large number of times and the best solution found is returned. Pardalos et al. have adapted *GRASP* to the FVSP in Pardalos et al. (1999) and Festa et al. (2009). The principle of the greedy randomized construction procedure used in this *GRASP* algorithm is as follows. Starting from the whole graph (corresponding to an empty feedback set), a vertex is removed from the graph (equivalently, this vertex is introduced into the feedback set). Then, reduction is applied. This process is repeated until there are no more cycles in the graph. Finally, a refinement procedure is applied to the feedback set.

The vertex selected on each iteration is chosen according to a greedy function. Three different functions have been tested in Pardalos et al. (1999). The principle of these functions is to elect a vertex that has a large number of incoming and outgoing vertices. The best greedy function identified by the authors is the product of the incoming degree by the outgoing degree. To select a vertex, the algorithm first introduces into a restricted candidate list (RCL) all vertices whose greedy score is greater than R times the largest greedy function value—where $R (0 \leq R \leq 1)$ is the randomization parameter of the algorithm. Then, a vertex is chosen randomly from the RCL. Note that the reduction operations applied are those proposed in Levy and Low (1988). The goal of the refinement procedure is simply to remove redundant vertices from the feedback set: this is done by testing, for each vertex in the feedback set, if it can be removed (i.e., reintroduced into the graph) without creating cycles.

Experiements performed with *GRASP* on two data sets are reported in Pardalos et al. 1999. The first data set contains nine very small graphs (having 25–35 vertices) whose optimum is known. *GRASP* found optimal solutions for all these graphs within very few *GRASP* iterations. The second data set contains 40 larger graphs (from 50 to 1,000 vertices) constructed by the authors. More details about these experiments will be given in Sect. 5.6.

3 A local search approach for the FVSP

In this section, we describe the local search approach we propose for the FVSP. We first define the search space (the set of configurations) and the evaluation function. Then, we present the move mechanism—equivalent to a neighborhood function. Finally, we propose a candidate list—equivalent to a restricted neighborhood.

In the following sections, we consider a directed graph $G = (V, E)$, with a set V of vertices and a set $E \subseteq V \times V$ of arcs. Given a subset $V' \subseteq V$ of the vertex set, we recall that the subgraph of G induced by V', denoted by $G(V')$, is the graph whose vertex set is V' and whose arcs are the arcs in E having their two endpoints in V': $G(V') = (V', E \cap (V' \times V'))$.

3.1 Solution representation

A directed graph with no directed cycles is named a *directed acyclic graph* (DAG). Every DAG has a *topological ordering*, i.e. an ordering of its vertices such that the starting-point of every arc occurs earlier in the ordering than the endpoint of the arc. Conversely, the existence of a topological ordering in a graph proves that this graph is acyclic. Note that the problem (named *topological sorting*) of finding a topological ordering of a given graph can be solved in $O(m + n)$.

Given a set $V' \subseteq V$ of vertices, we can observe that the induced subgraph $G(V')$ is acyclic if, and only if, $V - V'$ is a feedback set. Therefore, the FVSP is equivalent to finding a set V' of maximum cardinality such that $G(V')$ is acyclic. In the local search approach we propose, a configuration is equivalent to an acyclic induced subgraph. However, this induced subgraph will not be represented by a mere set of vertices but rather by one of its topological orderings.

3.2 Search space and evaluation function

In the proposed local search approach, a *configuration* is any (ordered) sequence of vertices such that, if the two endpoints of an arc belong to the sequence, the starting-point appears earlier than the endpoint. In a sequence S, we denote by $|S|$ the number of elements and by $S[i]$ the i-th element, for every $i = 1 \dots p$ where $p = |S|$. The sequence $S = (S[1], S[2] \dots S[p])$ is a (legal) configuration if:

1. $S[1], S[2] \dots S[p]$ belong to V and are all different;
2. $\forall i, j, (1 \le i < j \le p) \Rightarrow (S[j], S[i]) \notin E$.

Condition 1 indicates that S is a valid sequence. Condition 2 expresses the *precedence constraint* between the vertices present in the sequence. In the following, we will denote by $Dom(S) = \{S[1], S[2] \dots S[p]\}$ the set of the vertices that appear in the sequence. In addition, a vertex in V will be said numbered or unnumbered whether it belongs to $Dom(S)$ or not.

We can notice that the following *fundamental property* holds: If S is a configuration, then $G(Dom(S))$ is acyclic (and $V - Dom(S)$ is a feedback set). Therefore, every configuration corresponds actually to a feedback set.

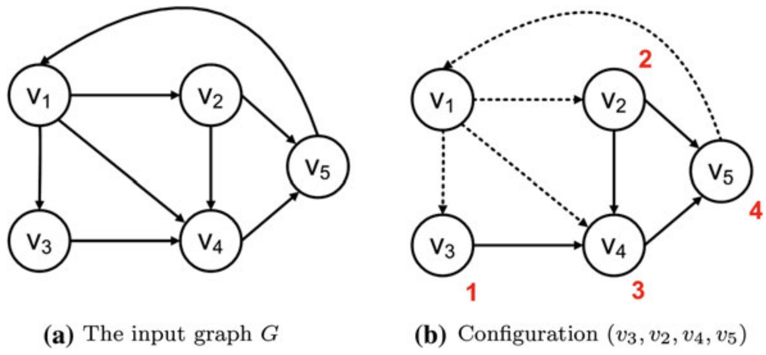

Fig. 1 Illustration of a configuration

The *evaluation function* f (to be minimized) is defined as follows. For any configuration S, $f(S)$ is the cardinality of the feedback set $V - Dom(S)$: $f(S) = |V - Dom(S)|$.

These definitions are illustrated in Fig. 1. A directed graph $G(V, E)$ is displayed on the left (Fig. 1a) and a configuration $S = (v_3, v_2, v_4, v_5)$ on the right (Fig. 1b). $Dom(S)$ equals $\{v_2, v_3, v_4, v_5\}$. The (acyclic) graph $G(Dom(S))$ is represented in plain lines. The cost of configuration S is $f(S) = |V - Dom(S)| = |\{v_1\}| = 1$.

3.3 Move mechanism

In the following, we consider a reference configuration S. A move consists in inserting a new vertex at some particular position into the sequence and, at the same time, in removing the vertices that would now violate the precedence constraint. More formally, given an unnumbered vertex $v \in V - Dom(S)$ and an integer $i \in \{1 \dots |S| + 1\}$, move $<v, i>$ consists in inserting v just before the element $S[i]$, and in removing from S the elements of $CV_-(v, i) \cup CV_+(v, i)$, where:

- $CV_-(v, i) = \{S[j] \in S : j \geq i \text{ and } (S[j], v) \in E\}$
- $CV_+(v, i) = \{S[j] \in S : j < i \text{ and } (v, S[j]) \in E\}$

We can notice that the so-obtained sequence is still a (legal) configuration. This configuration will be denoted by $S\oplus <v, i>$. According to our move definition, each unnumbered vertex can be inserted in $|S| + 1$ possible positions. Therefore, the number of moves applicable to a configuration (i.e., the size of the neighborhood) is $O(n^2)$. Note that the proposed move mechanism has some resemblance with the i-swap moves proposed for graph k-coloring in Morgenstern (1996).

The definition of a move is illustrated in Fig. 2, where move $<v_5, 3>$ is applied to configuration $S = (v_1, v_2, v_4)$. Configuration S is represented in Fig. 2a, where each numbered vertex $S[i]$ has a label equal to i. In 2b, vertex v is assigned a label equal to $3 - \epsilon$. In 2c, the labels of conflicting vertices ($CV_-(v_5, 3) = \{v_4\}$ and $CV_+(v_5, 3) = \{v_1\}$) are removed. Finally, the labeled vertices are renumbered in 2d. Thus, $S \oplus <v_5, 3> = (v_2, v_5)$.

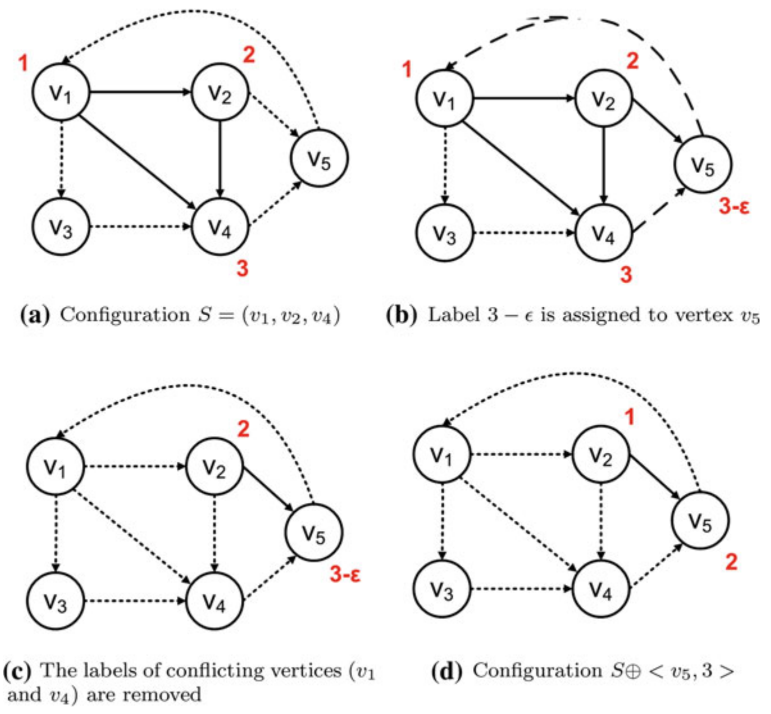
1
2
3
V1
V2
V3
V4
V5
3-ε
(a) Configuration $S = (v_1, v_2, v_4)$
(b) Label $3 - \epsilon$ is assigned to vertex v_5
(c) The labels of conflicting vertices (v_1 and v_4) are removed
(d) Configuration $S \oplus < v_5, 3 >$

Fig. 2 Applying move <v_5, 3> to configuration $S = (v_1, v_2, v_4)$

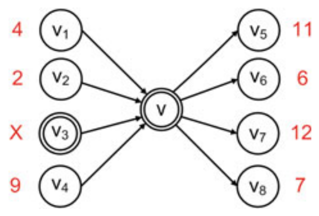

Fig. 3 A vertex v with its in-coming and out-going neighbors

In addition, we will denote by $\delta(v, i) = f(S \oplus <v, i>) - f(S)$ the performance of move $<v, i>$, i.e. its impact on the evaluation function. As a single vertex is introduced into the sequence (namely vertex v) and the vertices in $CV_-(v, i)$ and $CV_+(v, i)$ are removed, we have that

$$\delta(v, i) = -1 + |CV_+(v, i)| + |CV_-(v, i)|.$$

The calculation of the performance of a move is illustrated in Fig. 3. This figure shows a vertex v displayed along with its neighbors. Unnumbered vertices (v and v_3) are represented by doubled circles. Numbered vertices are represented with their index. Let us consider move $m = <v, 8>$. Conflicting vertices are those in $CV_-(m) = \{v_4\}$ and $CV_+(m) = \{v_6, v_8\}$. The performance of move m is $\delta(m) = -1 + |CV_-(m)| + |CV_+(m)| = -1 + 1 + 2 = 2$.

3.4 Candidate list strategy

A common strategy consists in defining a reduced set of moves (named a candidate list) that contains a subset of moves of high quality. The candidate list is intended to be exploited in a local search heuristic instead of the original set of moves in order to improve the efficiency of the search. In the following, we first describe the candidate list strategy we propose for the FVSP before discussing its potential merits.

According to our candidate list strategy, a vertex v can be inserted in the sequence in only two different positions : just after its numbered in-coming neighbors [in a position named $i_-(v)$], or just before its numbered out-going neighbors [in a position named $i_+(v)$]. Formally, these two positions are defined as follows:

- Let $I_-(v) = \{i : S[i] \in N_-(v)\}$ and $I_+(v) = \{i : S[i] \in N_+(v)\}$.
- $i_-(v) = \max(I_-(v)) + 1$ if $I_-(v) \neq \phi$; otherwise, $i_-(v) = 1$, and
- $i_+(v) = \min(I_+(v))$ if $I_+(v) \neq \phi$; otherwise, $i_+(v) = |S| + 1$.

where $N_-(v)$ and $N_+(v)$ denote the sets of the in-coming and out-going neighbors of vertex v, respectively.

For illustrating the calculation of $i_-(v)$ and $i_+(v)$, we consider again Fig. 3. $I_-(v) = \{2, 4, 9\}$ and $I + (v) = \{6, 7, 11, 12\}$ are the indexes of (numbered) in-coming and out-going neighbors of v, respectively. We have $i_-(v) = \max(I_-(v)) + 1 = 10$ and $i_+(v) = \min(I_+(v)) = 6$.

Let us now give comments about the candidate list. We first notice that the size of the candidate list is much smaller than the size of the whole set of moves: $O(n)$ versus $O(n^2)$. In addition, we have seen just above that applying move $<v, i_+(v)>$ means that vertex v is inserted into the sequence just before its out-going neighbors. As v is inserted before the out-going neighbors, this move will create no conflicts with these neighbors. Moreover, as v is inserted *just before* the out-going neighbors, it will create at the same time as few conflicts as possible with the in-coming neighbors. The same can be said, *mutatis mutandis*, about move $<v, i_-(v)>$. Therefore, the two moves $<v, i_+(v)>$ and $<v, i_-(v)>$ have been designed carefully in order to generate a limited number of conflicts. Two properties of the candidate list will be presented in the following section—see properties 2 and 3 in Sect. 3.5.

3.5 Properties

In this section, we present three important properties related to the whole set of moves and to the candidate list.

Property 1 *If a configuration S is a local optimum (with respect to the whole set of moves), this does not imply that the feedback set V-Dom(S) is minimal*

This point is illustrated in Fig. 4. The figure displays a graph along with a configuration $S = (v_3, v_1)$. This configuration is a local optimum because none of the

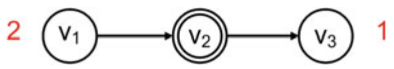

Fig. 4 A local optimum may not correspond to a minimal feedback set

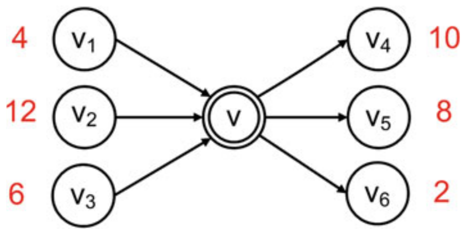

Fig. 5 The candidate list may not contain the best move applied to a vertex

three moves applicable to S is strictly improving—as numbering v_2 will necessarily unassign at least one other vertex. However $\{v_2\}$ is not a minimal feedback set: as the graph is acyclic, we have that the empty set is a feedback set strictly included in $\{v_2\}$.

Property 2 *Given an unnumbered vertex v, the candidate list may not contain the best move applicable to v*

This property is illustrated in Fig. 5. In this example, $\delta(v, i_-(v)) = \delta(v, 13) = 2$, $\delta(v, i_+(v)) = \delta(v, 2) = 2$, while $\delta(v, 7) = -1 + |\{v_2\}| + |\{v_6\}| = 1$. Thus, move $<v, 7>$ is strictly better than both $<v, i_-(v)>$ and $<v, i_+(v)>$.

Property 3 *A local optimum for the candidate list can not be improved by applying a move present in the whole set of moves: both sets of moves define the same local optima*

If there exists an improving move $<v, i>$ in the whole neighborhood ($\delta(v, i) = -1$), we have that $\max I_-(v) < \min I_+(v)$. Thus $\delta(v, i_-(v)) = \delta(v, i_+(v)) = -1$ are also improving moves.

4 A simulated annealing algorithm for the FVSP

Building on the local search approach described in the precedent sections, we have adapted the simulated annealing metaheuristic (Cerny 1985; Kirkpatrick et al. 1983) to the FVSP. Our Simulated Annealing algorithm for the Feedback Vertex Set Problem will be denoted by *SA-FVSP* in the following.

4.1 High-level description of the *SA-FVSP* algorithm

The pseudo-code of the algorithm is as follows:

```
Algorithm SA-FVSP()

Input a directed graph $G$
Parameters $T_0$, $\alpha$, maxMv, maxFail
1.  Set $T := T_0$; nbFail := 0; $S := ()$; $S* := ()$;
2.  Repeat
3.      Set nbMvt := 0; failure := true;
4.      Repeat
5.          Choose a move $<v, b>$ at random in the candidate list;
```

```
6.            Evaluate $\Delta := \delta(v, b)$;
7.            **If** $\Delta \leq 0$ or $\exp(-\Delta/T) \geq$ r and() **then**
8.                Apply move $<v, b>$ to configuration $S$;
9.                Set $nbMvt := nbMvt + 1$
10.               **If** $f(S) < f(S*)$ **then** Set $S* := S; failure := false$
11.       **Until** $nbMvt = maxMvt$;
12.       **If** $failure = true$ **then**
13.           Set $nbFail := nbFail + 1$
14.       **else**
15.           Set $nbFail := 0$
16.       Set $T := T \times \alpha$
17.   **Until** $nbFail = maxFail$
18.   **Return** $S*$
```

The *SA-FVSP* algorithm is a classical simulated annealing with a geometric cooling schedule. The initial temperature is fixed by a parameter (T_0). The initial configuration is an empty list of vertices. The pseudo-code contains two loops. An inner loop (lines 5–10) corresponds to a trial performed by the simulated annealing algorithm: a move is chosen randomly in the candidate list (line 5); then, the criterion of Metropolis determines if the move is accepted (line 7); if it is the case, the move is applied to the current configuration (lines 8–10). An outer loop (lines 3–16) corresponds to a *stage* of the algorithm. During a stage, the algorithm performs a series of inner loops (trials) until *maxMvt* "actual" moves have been performed; then the temperature is decreased (line 16). This process is repeated until *maxFail* stages have been performed without improvement of the score of the best configuration. Finally, the best configuration generated during the search is returned by the algorithm. We can notice that the algorithm is governed by the four following parameters:

- T_0 is the initial temperature;
- *maxMvt* is the number of moves performed during a stage;
- α is the factor used to decrease the temperature;
- *maxFail* is the number of stages performed without any improvement of the score of the best configuration.

4.2 Sketch of the implementation

In this section, we present and discuss complexity issues about the algorithm. We first sketch the implementation and then evaluate the complexity of a procedure that performs a local search iteration by exploring the whole candidate list and by evaluating each move.

For each vertex v, the algorithm stores two integers that contain the values of $\delta(v, i_+(v))$ and $\delta(v, i_-(v))$. It also stores a boolean $ok[v]$, a flag whose role is to indicate that the current values of $\delta(v, i_+(v))$ and $\delta(v, i_-(v))$ are valid. The flag of each vertex is set to false at the beginning of the search. A local search iteration is performed in two consecutive steps. During step 1, all vertices are scanned, the performance of each move is determined, and the best move $m = <v_m, i_m>$ is chosen. For each numbered vertex v such that $ok[v] = false$, a procedure named $UpdateVertex(v)$ is called. This procedure scans a first time $N_+(v)$ and $N_-(v)$ in order to compute $i_+(v)$ and $i_-(v)$, and then a second time in order to compute $\delta(v, i_+(v))$ and $\delta(v, i_-(v))$; then $ok[v]$ is set to $true$. Therefore, the complexity of the $UpdateVertex()$ procedure is $O(d_{max})$, where d_{max} denotes the maximum degree of a vertex in the graph.

During step 2, move $<v_m, i_m>$ is applied and the new configuration is built. Let us denote by C the set of vertices removed from the sequence. At the end of step 2, for every vertex x in $C \cup \{v_m\}$, and for every neighbor y of x in the graph, $ok[y]$ is set to $false$.

We can notice that the average number of vertices present in C per iteration (since the beginning of the search) is smaller or equal to 1—because the initial configuration is empty and exactly one vertex is inserted into the sequence on each iteration. Therefore, $O(d_{max})$ vertices in average have their flags set to $false$ during step 2. As procedure $UpdateVertex(.)$ is $O(d_{max})$ and as it is called $O(d_{max})$ times during step 1, the complexity of step 1 is $O(n + d_{max}^2)$. It is also the complexity of a whole local search iteration.

In the simulated annealing algorithm, as mentioned above in Sect. 4.1, a move is performed after performing a series of trials. A trial consists in generating a move $<v, b>$, evaluating the move and applying the metropolis criterion. Generating a move consists in choosing at random an unnumbered vertex v and then $b =$"+" or "−" with equal probability. In order to evaluate $<v, b>$ we simply read the stored value if $ok[v] = true$; otherwise, we first call procedure $UpdateVertex(v)$. Note that, when using the simulated annealing algorithm, performing an "actual" move necessitates an unbounded number of trials, and has therefore an unbounded complexity.

5 Computational results

In this section, we will present experiments conducted in order to evaluate the performance of our *SA-FVSP* algorithm (along with two variants of the algorithm) and to compare the obtained results to those of the *GRASP* algorithm (Pardalos et al. 1999).

The benchmark graphs used in our experiments are the graphs generated by Pardalos et al. (1999). Given the number n of vertices and m of arcs, a graph is constructed by choosing randomly m pairs of vertices. There are four subsets of graphs whose order equal 50, 100, 500 and 1,000 vertices, with 10 graphs in each subset. We have downloaded these graphs from http://www.research.att.com/~mgcr/.

The computer used in all our experiments is an Intel(R) Core(TM)2 CPU T8300 2.4 GHz with 2 GB of RAM.

5.1 Sketch of the experiments

In the following of this section, the *SA-FVSP* algorithm will be simply denoted by *SA*. This algorithm is the simulated annealing presented above. It uses the candidate list and the implementation described in Sect. 3.4. In our experiments, the *SA* algorithm will be compared to three other algorithms:

- the *SA* algorithm applied after reducing the input graph (denoted by *Red+SA*);
- the variant of the *SA* algorithm that uses the whole neighborhood (denoted by *SA-W*);
- the *GRASP* algorithm.

The *Red+SA* performs two successive stages. During the first stage, a reduction procedure is applied to the input graph in order to try to decrease the size of the graph. Then, the simulated annealing algorithm is applied to the reduced graph. We can assume that decreasing the size of the graph will render the problem easier and make it possible for the simulated annealing algorithm to reach better results. Our experiments with *Red+SA* (reported below in Sect. 5.4) will allow us to verify whether it is the case, and to measure to what extent.

The *SA-W* algorithm is similar to the *SA* algorithm, except that it uses the whole neighborhood instead of the candidate list. Note that the implementation described in Sect. 4.2 was not applicable to *SA-W*. Therefore, we have developed a specific implementation for *SA-W*. As the candidate list is a subset of the whole neighborhood, using it is likely to accelerate the algorithm, i.e. to render each local search iteration faster. On the other hand, as some good-performing moves are absent from the candidate list (see Sect. 3.4), using the candidate list may hamper the efficiency of the algorithm. Our experiments with *SA-W* (reported below in Sect. 5.5) are intended to shed light on that point.

Finally, we will compare the results of our *SA* algorithm to those of the *GRASP* algorithm by Pardalos et al. (1999). Experiments performed with this algorithm are reported in Pardalos et al. (1999). However, in these experiments, the *GRASP* algorithm was run only once and for only 100 iterations. In addition, it is difficult to have a clear idea about the relative speed between their computer and ours. Fortunately, the authors provide the source code of their algorithm making it possible to perform extended experiments on our computer. These experiments (reported below in Sects. 5.6 and 5.7) will allow us to realize a fair and extensive comparison between *SA* and *GRASP*.

5.2 Setting the parameters

The values of the parameters used by the four algorithms are given in Table 1. In the experiments performed with *GRASP*, we use the same parameters as the authors ($R = 0.8$, *maxGrIter*=100). With these parameters, performing on our computer one run with the *GRASP* algorithm takes between less than one second and up to about 800 seconds, depending on the graph.

For the *SA* algorithm, we have fixed the values of T_0 and *maxFail* after a limited number of preliminary experiments. The value of parameter α was chosen arbitrarily (a little bit smaller than 1); then we have fixed the value of *maxMvt* so as to obtain cpu times that are roughly comparable to those of *GRASP*. Note that multiplying the value of *maxMvt* by some coefficient $p > 1$ has the effect to roughly multiply cpu times by p, while improving more or less the results, depending on p. With the chosen parameter setting, the computing times (per run) of our *SA* algorithm range from less than one second to almost 30 seconds, depending on the graph, and they are generally

Table 1 Parameter settings used during the experiments

Algorithm	Parameter	Value	
SA	T_0	0.6	Initial temperature
Red+SA	*maxMvt*	$5 \times n$	Number of moves performed during each stage
SA-W	α	0.99	Factor used to decrease the temperature
	maxFail	50	Number of stages without improvement
GRASP	*R*	0.8	Randomization parameter
	maxGrIter	100	Number of iterations

equal to or lower than those of *GRASP*. Therefore, comparing one run of *SA* to one run of *GRASP* should never disadvantage *GRASP*—all the contrary.

In our experiments, we have performed with the four algorithms a series of 30 runs on each of the 40 benchmark graphs. In addition, we have conducted extended tests with *SA* and *GRASP* while using much larger computing times. In these extended experiments, *SA* was run 1,000 times (versus 30 times in the regular experiment) and *GRASP* was run once for as much as 20,000 iterations (versus a total of $30 \times 100 = 3{,}000$ iterations during the regular experiment).

5.3 Results obtained by the *SA* algorithm

Table 2 displays the results obtained during the experiments by the *SA* algorithm. Each line in Table 2 corresponds to a graph identified by its number n of vertices and its number m of arcs (in column 1 and 2). Columns 3–6 display statistics (minimum, average, maximum and standard deviation) about the size of the feedback set returned by *SA*, over the 30 runs. Columns 7 and 8 give the total time and the total number of iterations per run, averaged over the 30 runs—the total time is the time measured when the algorithm stops, not just the time used to reach the best solution. Column 9 (labeled ”Best SA”) gives the size of the smallest feedback set found during the extended experiment of 1,000 runs. The last column (labeled ”Diff”) indicates the difference between the best result found during the 30 runs and the best result found during the extended test.

From the table, we can observe that computing times of *SA* range between 0.03 and 0.07 s for $n = 50$, 0.08 and 0.34 s for $n = 100$, 1.8 and 5.2 s for $n = 500$, and 11 and 25.5 s for $n = 1{,}000$.

Let us observe the value of column 10 (”Diff”) along with the one of the standard deviation of f. For small graphs, we notice that these two values are low. This may indicate that the algorithm finds on each run a solution equal or close to the optimum—although we can not be sure of the value of the optimum. The opposite is true for

Table 2 Results obtained by *SA* on the benchmark graphs

Graph		SA						Best*SA*	Diff
		f				cpuT	iterT	f	f
n	m	min	avg	max	dev	avg	avg	min	min
50	100	3	3	3	0	0.03	12,875	3	0
50	150	9	9	9	0	0.03	13,600	9	0
50	200	13	13	13	0	0.03	13,817	13	0
50	250	17	17	17	0	0.03	14,125	17	0
50	300	19	19	19	0	0.04	13,658	19	0
50	500	28	28	28	0	0.05	18,200	28	0
50	600	31	31.4	32	0.5	0.07	23, 633	31	0
50	700	33	33	33	0	0.05	17,533	33	0
50	800	34	34.1	35	0.3	0.07	23, 558	34	0
50	900	36	36	36	0	0.04	17,800	36	0
100	200	9	9.1	10	0.3	0.08	33,083	9	0
100	300	17	17	17	0	0.1	35,933	17	0
100	400	23	23	24	0.2	0.11	38,667	23	0
100	500	32	32.3	33	0.4	0.16	47,400	32	0
100	600	37	37	37	0	0.16	47,000	36	1
100	1,000	53	53.2	54	0.4	0.28	53,883	53	0
100	1,100	54	54.8	55	0.4	0.23	43,083	54	0
100	1,200	57	57	58	0.2	0.29	52,233	57	0
100	1,300	60	6	61	0.2	0.31	52,817	60	0
100	1,400	61	61	61	0	0.34	48,833	61	0
500	1,000	31	32.1	33	0.7	1.79	236,500	31	0
500	1,500	64	65.1	66	0.6	2.18	276,667	64	0
500	2,000	102	104	106	0.9	2.61	324,167	102	0
500	2,500	133	135.5	138	1.1	2.75	327,333	133	0
500	3,000	164	165.4	168	1	2.76	312,583	162	2
500	5,000	237	239.2	241	1.2	3.78	298,917	237	0
500	5,500	252	253.8	256	1.3	3.96	305,333	251	1
500	6,000	265	267.6	270	1.3	4.64	324,750	264	1
500	6,500	277	278.9	283	1.2	4.79	319,833	276	1
500	7,000	287	288.9	292	1.2	5.2	324,667	286	1
1,000	3,000	132	134.3	137	1.3	11.5	715,167	129	3
1,000	3,500	166	168.8	172	1.3	11.3	718,833	164	2
1,000	4,000	196	198.9	202	1.6	11.49	727,500	195	1
1,000	4,500	229	234.2	239	2.6	10.98	698,000	228	1
1,000	5,000	263	265.6	269	1.3	11.4	718,000	259	4
1,000	10,000	472	475.4	479	1.5	13.45	730,167	469	3

Table 2 continued

Graph		SA						Best*SA*	Diff
		f				cpuT	iterT	f	f
n	m	min	avg	max	dev	avg	avg	min	min
1,000	15,000	582	584.9	588	1.8	16.73	694,500	579	3
1,000	20, 000	652	656.1	660	1.6	20.29	686, 833	651	1
1,000	25, 000	701	704.5	707	1.4	24.76	671, 667	699	2
1,000	30, 000	741	744	747	1.7	25.47	637, 500	739	2

the larger graphs. In particular, for graphs of 1,000 vertices, the value of column 10 ("Diff") ranges between 1 and 4. This confirms that the algorithm was never able to reach the optimal value and sometimes found solutions far from the optimum. However, this is not so surprising as the computing times are low, considering the size of the graph. Better and more robust results could be obtained by allotting more time to the algorithm, simply by increasing the value of parameter *maxMvt*—or even by performing restarts and returning the best solution.

When considering the number of moves, it must not be forgotten that the values displayed in the table correspond to the total number of moves carried out during one run. However, after finding the best solution, the algorithm still performed at least $maxFail \times maxMvt = 50 \times 5 \times n = 250 \times n$ moves. For example, if we consider the smallest graph, it took in average at most $12{,}875 - 250 \times 50 = 375$ moves (more precisely, between 125 and 375 moves) to reach the best solution (i.e. a feedback set of size 3). For the same reason, the computing time used to reach the best solution may be much smaller than the total cpu time indicated in the table, in particular for small graphs.

5.4 Tests performed with reduced graphs

Table 3 displays the results obtained by the *Red+SA* algorithm. Recall that *Red+SA* simply consists in applying a reduction procedure to the graph before using the simulated annealing algorithm. This reduction procedure performs recursively the five reduction operations proposed in Levy and Low (1988). It stops when no more reduction operation is applicable—note that the output graph does not depend on the order in which the operations are performed. This procedure is the same as the one used in *GRASP*—see Sect. 2.

Table 3 gives the results for only 23 graphs. For the 17 remaining graphs, the reduction procedure had no effect—the graph returned by the reduction procedure was the same as the input graph.

Columns 3 (labeled n') and 4 (m') give the number of vertices and arcs of the reduced graph, respectively. The other columns have the same meaning as in Table 2. The two last columns ("Diff") correspond to the difference between the results obtained by *Red+SA* and *SA*: therefore, a negative value indicates that the result of

Table 3 Results obtained by *Red+SA* on the benchmark graphs

Graph		Reduction		Red+SA						SA		Diff	
				f				cpuT	iterT	f		f	
n	m	n'	m'	min	avg	max	dev	avg	avg	min	avg	min	avg
50	100	11	36	3	3	3	0	0.03	12, 750	3	3	0	0
50	150	31	117	9	9	9	0	0.03	13, 075	9	9	0	0
50	200	41	180	13	13	13	0	0.03	13, 183	13	13	0	0
50	250	45	243	17	17	17	0	0.04	14, 600	17	17	0	0
50	300	47	297	19	19	19	0	0.04	13, 883	19	19	0	0
100	200	26	79	9	9	9	0	0.06	25, 900	9	9.1	0	−0.1
100	300	66	254	17	17	17	0	0.08	28, 900	17	17	0	0
100	400	73	320	23	23	23	0	0.1	36, 183	23	23	0	0
100	500	90	475	32	32.2	33	0.4	0.14	43, 933	32	32.3	0	−0.1
100	600	95	591	37	37.1	38	0.3	0.15	46, 066	37	37	0	0.1
100	1,000	99	998	53	53.2	54	0.4	0.27	53, 483	53	53.2	0	−0.1
500	1,000	122	407	31	31	31	0	1.78	228, 083	31	32.1	0	−1.1
500	1,500	309	1,224	63	64.3	65	0.5	2.35	288, 416	64	65.1	−1	−0.8
500	2,000	404	1,848	102	103.2	104	0.6	2.53	304, 750	102	104	0	−0.8
500	2,500	457	2,442	133	135.3	138	1.2	2.62	312, 000	133	135.5	0	−0.2
500	3,000	483	2,965	163	165.4	167	1.1	2.94	329, 166	164	165.4	−1	0
500	5,000	499	4,998	237	239.1	242	1.2	3.95	306, 583	237	239.2	0	−0.1
500	5,500	499	5,499	252	253.7	256	1.1	4.04	307, 416	252	253.8	0	−0.1
1,000	3,000	613	2,369	128	131.2	135	1.6	11.53	691, 666	132	134.3	−4	−3.1
1,000	3,500	723	3,026	163	166.5	169	1.5	12.34	753, 500	166	168.8	−3	−2.3
1,000	4,000	793	3,589	194	197.3	201	1.3	12.93	715, 500	196	198.9	−2	−1.6
1,000	4,500	869	4,231	230	233.5	237	2.3	12.21	748, 500	229	234.2	1	−0.7
1,000	5,000	919	4,839	263	265.7	269	1.3	11.7	731, 000	263	265.6	0	0.1

Red+SA is better than the one obtained by *SA* (thus, the reduction had a favourable impact in this case).

Let us first observe in columns 3–4 the outcome of the reduction. For a given value of n, the graphs that are actually reduced are those that have the smallest number of arcs. Moreover, the smallest the number of arcs, the most important the decrease in the number of vertices and arcs. We also notice that the average degree of a vertex tends to increase as a result of the reduction—this is not surprising as the reduction tends to remove the sparsest parts of the input graph. Finally, note that the computing time of the reduction is negligible.

Let us now analyse the impact of the reduction on the quality of the results. We notice that reduction makes it possible to improve significantly the results of some graphs with the minimum and the average reduced by up to 4 and 3.1 vertices, respectively. Unsurprisingly the improvement was the most important for the largest and sparsest graphs—those that underwent the most drastic shrinking. For three graphs of 1,000 vertices (those having 3,000, 3,500 and 4,000 arcs), the best solution found by $Red-$

SA within 30 runs (128, 163, and 194, respectively) is even better than the best solution found by SA within 1,000 runs (129, 164, and 195; see Table 2). Finally, we notice that $Red + SA$ is sometimes a little bit slower than SA. However, it is not so surprising if we consider that the reduction increased the average degree of a vertex.

In summary, when applied to a sparse graph, first reducing the input graph before applying local search may have a very positive impact on the quality of the solutions found.

5.5 Tests performed with the whole neighborhood

Table 4 displays the results obtained by the *SA-W* algorithm. The columns of the table have the same meaning as in the precedent tables. Positive values in the two last columns (labeled "Diff") indicate that the result of *SA* is better than the one obtained by *SA-W* (indicating that the candidate list has a favourable impact).

We can observe that the overall results of *SA-W* are not as good as those obtained by *SA*, with the minimum and the average increased by up to 19 and 20, respectively. The deterioration affects mainly the graphs having 500 and 1,000 vertices—the worst deterioration arising for the graphs of 1,000 vertices, except the densest ones.

In our experiments, the cpu times of *SA-W* (not reported in the table) were between 4 and 8 times larger than those of *SA*. Whatever the implementation of the *SA-W* algorithm, it is clear that it will be possible to make *SA* as fast as *SA-W*, and probably faster. However, as *SA* outperforms *SA-W*, we did not put too much effort in trying to optimize the implementation of *SA-W*. This is why we do not report the computing times of *SA-W* in Table 4.

As shown in Sect. 3.4, we are able to compute very efficiently the performance of a move chosen in the candidate list. It was therefore expected that computing an iteration was faster when using the candidate list than with the whole neighborhood. However, it is surprising to observe that restricting the set of moves to the candidate list does not affect the efficiency of the search—with respect to the decrease of the cost function for a same number of iterations. Far from that, the efficiency of the search is consistently and significantly improved. In other words, using the candidate list seems to better guide the search.

In summary, our experiments show that the simulated annealing algorithm is both faster and more efficient when using the candidate list rather than the whole neighborhood.

5.6 Replicating the results of *GRASP*

The source code of *GRASP* is provided by the authors of Pardalos et al. (1999) and can be downloaded from http://www.research.att.com/~mgcr/. Information about how to use it is provided in Festa et al. (2001). We have compiled this source code (written in Fortran). Before comparing in the next section the results obtained with this program to those of our SA algorithm, we first compare them to those reported in the original paper (Pardalos et al. 1999).

Table 4 Results obtained by *SA-W* on the benchmark graphs

Graph		SA-W				SA		Diff	
		f				f		f	
n	m	min	avg	max	dev	min	avg	min	avg
50	100	3	3	3	0	3	3	0	0
50	150	9	9	9	0	9	9	0	0
50	200	13	13.1	14	0.3	13	13	0	0.1
50	250	17	17	17	0	17	17	0	0
50	300	19	19	19	0	19	19	0	0
50	500	28	28	28	0	28	28	0	0
50	600	31	31.1	32	0.3	31	31.4	0	−0.3
50	700	33	33	33	0	33	33	0	0
50	800	34	34	34	0	34	34.1	0	−0.1
50	900	36	36	36	0	36	36	0	0
100	200	9	10.1	12	0.5	9	9.1	0	1
100	300	17	18.5	20	0.8	17	17	0	1.5
100	400	23	23.4	24	0.5	23	23	0	0.3
100	500	32	32.7	33	0.5	32	32.3	0	0.4
100	600	37	37.9	39	0.6	37	37	0	0.9
100	1,000	53	53.7	55	0.6	53	53.2	0	0.5
100	1,100	55	55	55	0	54	54.8	1	0.2
100	1,200	57	57.7	58	0.5	57	57	0	0.7
100	1,300	60	60	60	0	60	60	0	0
100	1,400	61	61	61	0	61	61	0	0
500	1,000	37	39.7	42	1.2	31	32.1	6	7.6
500	1,500	70	75.4	79	2.1	64	65.1	6	10.3
500	2,000	109	112.7	115	1.4	102	104	7	8.7
500	2,500	141	144.3	147	1.3	133	135.5	8	8.7
500	3,000	168	170.5	174	1.6	164	165.4	4	5.1
500	5,000	241	244.4	247	1.5	237	239.2	4	5.2
500	5,500	259	260.4	263	1.1	252	253.8	7	6.6
500	6,000	269	272.5	275	1.6	265	267.6	4	4.9
500	6,500	281	284.4	287	2	277	278.9	4	5.5
500	7,000	291	294.2	297	1.7	287	288.9	4	5.2
1,000	3,000	147	153.1	157	2.7	132	134.3	15	18.8
1,000	3,500	180	185	190	2.3	166	168.8	14	16.1
1,000	4,000	214	218	222	1.9	196	198.9	18	19.1
1,000	4,500	248	254.3	260	3	229	234.2	19	20
1,000	5,000	276	285.1	290	2.8	263	265.6	13	19.6
1,000	10,000	489	491.9	495	1.7	472	475.4	17	16.5
1,000	15,000	598	600.4	604	1.7	582	584.9	16	15.5
1,000	20,000	665	668	671	1.7	652	656.1	13	12

Table 4 continued

Graph		SA-W				SA		Diff	
		f				f		f	
n	m	min	avg	max	dev	min	avg	min	avg
1,000	25,000	710	714.5	718	2	701	704.5	9	10
1,000	30,000	748	752.3	755	1.8	741	744	7	8.3

Table 5 reproduces in columns 3 and 4 (labeled "Pardalos et al. 1999") the results given in Pardalos et al. (1999). Recall that the *GRASP* algorithm was run only once by the authors. Columns 5–9 (labeled "GRASP") display statistics about the results obtained by *GRASP* on our computer, over 30 runs. The size of the smallest feedback sets returned by *GRASP* during the extended experiment are presented in column 10 (labeled "BG", where "BG" stands for "Best GRASP").

We notice that, for each graph, the result reached by the authors during their single run is close to the average value obtained in our experiments; in addition, for every graph, this result falls between the minimum and the maximum value obtained on our computer. This confirms that the source code provided by the authors is the same or is equivalent to the one they used in their experiments.

When observing computing times, we notice unsurprisingly that those measured on our computer are shorter than those reported by the authors. Depending on the graph, the speeding rate generally ranges between 8 and 20. It seems to be very low, considering the speed up of computers since that time. However, this may not be so surprising when we consider that the computer used in Pardalos et al. (1999) (a Silicon Graphics Challenge computer with twenty 196 MHz MIPS R10000 processors and 6.1 Gb of main memory) is a "supercomputer" that was one of the fastest computers of that time (The New York Times 1993; http://en.wikipedia.org/wiki/SGI_Challenge).

5.7 Comparing the results of *SA* to those of *GRASP*

The two last columns ("Diff") in Table 5 correspond to the difference between the results obtained by *GRASP* and *SA*: therefore, a positive value indicates that the result of *SA* is better than the one obtained by *GRASP*.

If we compare *SA* and *GRASP* with respect to solution quality, we notice that, for every graph, the results obtained by *GRASP* are never better than those of SA, whether we consider the minimum or the average. The improvement of *SA* over *GRASP*, according to the minimum, range between 0 and 1, 0 and 3, 1 and 26, and 10 and 55 for graphs of 50, 100, 500 and 1,000 vertices, respectively. These results indicate therefore a moderate advantage for *SA* on the graphs with 50 vertices, but an important advantage for the graphs of 100 vertices, and an considerable difference for the graphs having 500 and 1,000 vertices.

Let us consider the size of the smallest feedback set found by *GRASP* during a run of 20,000 iterations (column labeled "BG"). In this experiment, *GRASP* was allotted 200 more times than during a regular run. In spite of that, the result of *GRASP* is

Table 5 Comparative results obtained by *GRASP* and *SA* on the benchmark graphs

Graph		Pardalos et al. (1999)		GRASP					BG	SA			Diff	
		f	cpuT	f				cpuT	f	f		cpuT	f	
n	m			min	avg	max	dev	avg	min	min	avg	cpuT	min	avg
50	100	3	0.1	3	3	3	0	0.01	3	3	3	0.03	0	0
50	150	9	0.3	9	9	9	0	0.03	9	9	9	0.03	0	0
50	200	13	0.5	13	13	13	0	0.05	13	13	13	0.03	0	0
50	250	17	0.65	17	17	18	0.2	0.08	17	17	17	0.03	0	0
50	300	19	0.7	19	19.1	22	0.5	0.1	19	19	19	0.04	0	0.1
50	500	29	1.35	29	29	30	0.2	0.18	28	28	28	0.05	1	1
50	600	32	2.21	32	32.6	35	0.7	0.27	32	31	31.4	0.07	1	1.2
50	700	33	2	33	33	33	0	0.29	33	33	33	0.05	0	0
50	800	36	2.98	35	35.8	38	0.6	0.36	35	34	34.1	0.07	1	1.7
50	900	36	3.42	36	36.1	38	0.4	0.42	36	36	36	0.04	0	0.1
100	200	9	0.55	9	9	9	0	0.05	9	9	9.1	0.08	0	–0.1
100	300	17	1.8	17	17.1	19	0.4	0.15	17	17	17	0.1	0	0.1
100	400	23	1.57	23	23.1	25	0.4	0.2	23	23	23	0.11	0	0.1
100	500	33	2.94	33	33	34	0.2	0.35	32	32	32.3	0.16	1	0.8
100	600	39	3.81	38	38.9	42	0.8	0.46	38	37	37	0.16	1	1.9
100	1,000	56	7.23	55	55.8	58	0.6	0.88	54	53	53.2	0.28	2	2.6
100	1,100	58	7.49	57	58.2	62	0.9	0.95	56	54	54.8	0.23	3	3.4
100	1,200	61	8.2	59	60.4	64	0.9	1.11	59	57	57	0.29	2	3.4
100	1,300	63	9.92	62	63	69	1.2	1.26	62	60	60	0.31	2	2.9
100	1,400	64	11.06	62	63.5	68	1	1.38	62	61	61	0.34	1	2.5
500	1,000	34	10.07	32	33.2	35	0.8	1.33	32	31	32.1	1.79	1	1.1

Table 5 continued

Graph		Pardalos et al. (1999)		GRASP					BG	SA			Diff	
		f	cpuT	f				cpuT	f	f		cpuT	f	
n	m			min	avg	max	dev	avg	min	min	avg	cpuT	min	avg
500	1,500	71	29.92	69	70.8	74	0.9	2.4	67	64	65.1	2.18	5	5.7
500	2,000	111	64.62	109	111.5	119	1.6	7.37	108	102	104	2.61	7	7.5
500	2,500	152	86.17	152	153.7	159	1.3	5.4	150	133	135.5	2.75	19	18.2
500	3,000	183	97.66	180	182.5	190	1.7	12.13	180	164	165.4	2.76	16	17.1
500	5,000	262	205.13	260	262.3	268	1.7	23.03	258	237	239.2	3.78	23	23.1
500	5,500	281	233.28	278	281.1	291	2.1	15.25	274	252	253.8	3.96	26	27.3
500	6,000	292	270.95	289	293	300	1.7	29.45	288	265	267.6	4.64	24	25.4
500	6,500	304	308.96	299	304.1	313	2.3	19.39	299	277	278.9	4.79	22	25.2
500	7,000	317	345.2	313	315.7	322	1.5	35.39	309	287	288.9	5.2	26	26.7
1,000	3,000	148	126.96	142	147.2	155	2.1	15.73	143	132	134.3	11.5	10	12.9
1,000	3,500	185	176.65	180	183.5	189	1.7	12.24	179	166	168.8	11.3	14	14.7
1,000	4,000	217	219.8	216	218.3	223	1.7	25.97	213	196	198.9	11.49	20	19.4
1,000	4,500	261	264.63	257	261.1	272	2.4	19.27	254	229	234.2	10.98	28	26.9
1,000	5,000	299	310.48	294	298.6	311	2.6	36.1	294	263	265.6	11.4	31	33.1
1,000	10,000	531	1025.53	526	531.1	539	2.3	89.27	524	472	475.4	13.45	54	55.7
1,000	15,000	638	2171.69	634	638.8	641	1.7	156.14	629	582	584.9	16.73	52	53.9
1,000	20,000	713	3971.28	707	709.7	713	1.3	245.06	705	652	656.1	20.29	55	53.6
1,000	25,000	755	5117.78	751	754.1	760	1.8	234.35	747	701	704.5	24.76	50	49.7
1,000	30,000	792	7206.13	788	792.8	802	2.5	321.76	788	741	744	25.47	47	48.8

generally worse than the average result obtained by SA during a single regular run. For every graph of 1,000 vertices, the record of *GRASP* is much worse than the average result of SA; it is even worse than the worst result of SA obtained over the 30 runs.

In summary, while the performances of *SA* and *GRASP* are comparable on a few very small graphs, *SA* outperforms *GRASP* by a very large margin for large graphs of 500 and 1,000 vertices.

6 Conclusion

Although the FVSP is an important NP-hard problem, local search heuristics had never been applied to the problem, and no local search approach was known in order to tackle it. In this paper, we have proposed a practical local search approach for the solution of this problem.

Taking advantage of a well-known property related to acyclic graphs, we have proposed a new representation of feedback sets: given a graph $G(V, E)$, a feedback set V' is represented by a linear ordering of the subgraph induced by $V - V'$. Thanks to this solution representation, it becomes possible to define a move mechanism (equivalent to a neighborhood) that transforms a given feedback set into a new legal feedback set.

In addition, we have identified a reduced set of moves (i.e., a candidate list) that contains a subset of high-quality moves. The cardinality of the candidate list is much smaller than the one of the original set of moves (linear versus quadratic in $|V|$). We have also described an efficient technique to evaluate incrementally the performance of the moves of the candidate list.

We have implemented a simulated annealing algorithm that exploits the proposed local search approach and tested the algorithm on standard benchmark graphs. Our experiments show that using the candidate list instead of the whole neighborhood has a positive impact on the results. Above all, the experiments show that our algorithm outperforms by a large margin the best existing heuristic, namely the *GRASP* by Pardalos et al. Pardalos et al. (1999). These results demonstrate the efficiency of the proposed local search approach for the solution of the FVSP.

The local search approach proposed in this paper paves the way for new more powerful heuristics. In particular, a promising avenue is the development of hybrid heuristics, such as memetic algorithms. This will be the subject of our future work.

References

AT&T Labs: Personal page of Mauricio Resende at AT&T Labs. http://www2.research.att.com/~mgcr/. Accessed Mar 2013

Bafna, V., Berman, P., Fujito, T.: Approximating feedback vertex set for undirected graphs within ratio 2, manuscript (1994)

Becker, A., Geiger, G.: Approximation algorithms for the loop cutset problem. In: Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence, pp. 60–68 (1979)

Cerny, V.: Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. J. Optim. Theory Appl. **45**, 41–51 (1985)

Feo, T.A., Resende, M.G.C.: Greedy randomized adaptive search procedures. J. Glob. Optim. **6**, 109–133 (1995)

Festa, P., Pardalos, P.M., Resende, M.G.C.: Algorithm 815: FORTRAN subroutines for computing approximate solutions of feedback set problems using GRASP. ACM Trans. Math. Softw. **27**(4), 456–464 (2001)

Festa, P., Pardalos, P.M., Resende, M.G.C.: Feedback set problems. In: Floudas, C.A., Pardalos, P.M. (eds.) Encyclopedia of optimization, pp. 1005–1016. Springer, Heidelberg (2009)

Garey, M.R., Johnson, D.S.: Computers and reducibility: a guide to the theory of NP-completeness. W.H. Freeman, San Francisco (1979)

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)

Levy, H., Low, D.W.: A contraction algorithm for finding small cycle cutsets. J. Algorithms **9**(4), 470–493 (1988)

Lin, H.M., Jou, J.Y.: Computing minimum feedback vertex sets by contraction operations and its applications on CAD. In: (ICCD '99) International Conference on Computer Design, pp. 364–369 (1999)

Monien, B., Schultz, R.: Four approximation algorithms for the feedback vertex set problem. In: Proceedings of the 7th Conference on Graph Theoretic Concepts of Computer Science, pp. 315–390 (1981)

Morgenstern, C.: Distributed coloration neighborhood search. In: Johnson, D., Trick, M. (eds.) Cliques, coloring, and satisfiability, DIMACS: series in discrete mathematics and theoretical computer science, vol. 26, pp. 335–357. American Mathematical Society, New Providence (1996)

Pardalos, P.M., Qian, T., Resende, M.G.C.: A greedy randomized adaptive search procedure for feedback vertex set. J. Comb. Optim. **2**, 399–412 (1999)

Qian, T., Ye, Y., Pardalos, P.M.: A pseudo-approximation algorithm for FVS. In: Floudas, C., Pardalos, P. (eds.) State of the art in global optimization, pp. 341–351. Kluwer Academic Publishers, Dordrecht, Boston, London (1996)

Seymour, P.D.: Packing directed circuits fractionally. Combinatorica **15**, 182–188 (1995)

The New York Times: New 'micros' disclosed. The New York Times, 28 Jan 1993

Wikipedia. http://en.wikipedia.org/wiki/SGI_Challenge. Accessed Mar 2013

Yannakakis, M.: Node and edge-deletion NP-complete problems. In: Proceedings of the 10th Annual ACM Symposium on Theory of Computing, pp. 253–264 (1978)

Yehuda, B., Geiger, D., Naor, J., Roth, R.M.: Approximation algorithms for the vertex feedback set problem with applications to constraint satisfaction and Bayesian inference. In: Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 344–354 (1994)