



Delayed improvement local search

Heber F. Amaral^{1,2} · Sebastián Urrutia^{2,3}  · Lars M. Hvattum³

Received: 26 November 2020 / Revised: 21 May 2021 / Accepted: 25 May 2021 / Published online: 29 June 2021
© The Author(s) 2021

Abstract

Local search is a fundamental tool in the development of heuristic algorithms. A neighborhood operator takes a current solution and returns a set of similar solutions, denoted as neighbors. In best improvement local search, the best of the neighboring solutions replaces the current solution in each iteration. On the other hand, in first improvement local search, the neighborhood is only explored until any improving solution is found, which then replaces the current solution. In this work we propose a new strategy for local search that attempts to avoid low-quality local optima by selecting in each iteration the improving neighbor that has the fewest possible attributes in common with local optima. To this end, it uses inequalities previously used as optimality cuts in the context of integer linear programming. The novel method, referred to as delayed improvement local search, is implemented and evaluated using the travelling salesman problem with the 2-opt neighborhood and the max-cut problem with the 1-flip neighborhood as test cases. Computational results show that the new strategy, while slower, obtains better local optima compared to the traditional local search strategies. The comparison is favourable to the new strategy in experiments with fixed computation time or with a fixed target.

Keywords Combinatorial optimization · Local search inequalities · Heuristic · 2-Opt · Traveling salesman problem · Max-cut

✉ Sebastián Urrutia
sebastian.a.urrutia@himolde.no

Heber F. Amaral
heber.amaral@ifsudestemg.edu.br

Lars M. Hvattum
hvattum@himolde.no

¹ Instituto Federal do Sudeste de Minas Gerais, R. da Independência, 30 - Aparecida, Bom Sucesso, Minas Gerais, Brazil

² Universidade Federal de Minas Gerais, Av. Antônio Carlos, 6627, Pampulha, Belo Horizonte, Minas Gerais, Brazil

³ Faculty of Logistics, Molde University College, Britvegen 2, 6410 Molde, Norway

1 Introduction

Finding an optimal solution to a given instance of a combinatorial optimization problem can be a hard computational task. Most of the methods used to solve hard combinatorial optimization problems fall into two major groups: exact methods, whose goal is to find an optimal solution of the problem together with a proof of optimality, and heuristic methods, which seek to find a sufficiently good solution within a reasonable time limit. One of the most well-known heuristic techniques is called local search and is the focus of this article.

Local search procedures start from a feasible initial solution and proceed by applying operations on the current solution to obtain better solutions. The set of solutions that can be generated with a given operator on a solution s is called the neighborhood of s . Local search procedures choose, from the neighborhood of the current solution, one feasible solution that improves the value of the objective function. The chosen solution becomes the new current solution. This process is repeated until there are no improving solutions in the neighborhood of the current solution. A solution having no improving solution in its neighborhood is called a local optimum.

Strategies for implementing local search differ in the way the new solution is chosen among the improving solutions in the current neighborhood. This work proposes a delayed improvement local search (DILS) strategy, which attempts to avoid converging to a low-quality local optimum. This is achieved by first defining attributes that are present in all local optima, and then selecting at each iteration of the local search an improving neighbor that has the fewest possible attributes in common with local optima. The DILS is compared to the well-known best improvement local search (BILS) and first improvement local search (FILS) strategies.

To evaluate DILS we consider two different optimization problems and corresponding neighborhood operators. The first is the well-known travelling salesman problem (TSP) with the 2-opt neighborhood (Johnson and McGeoch 1997). This neighborhood is commonly used when developing more elaborate heuristic techniques for the TSP and related problems. The second test case is the max-cut problem, using the 1-flip neighborhood (Festa et al. 2002).

The main objective of this article is the proposal of a new strategy for local search procedures applied to combinatorial optimization problems. We suggest the use of local search inequalities (Lancia et al. 2015) for implementing our strategy. Finally, we evaluate the proposed strategy by comparing its behavior and performance to that of two classic local search strategies.

The rest of this work is organized as follows: Sect. 2 provides background material and a review of the related literature. In Sect. 3 we present the general form of DILS. Section 4 deals with the application of DILS to the TSP using the 2-opt neighborhood. Then, the application of DILS to max-cut using the 1-flip neighborhood is shown in Sect. 5. In Sect. 6 we explain how our computational experiments are structured. Computational results focus on demonstrating the behavior of DILS, as well as evaluating the performance. Section 7 presents results for the TSP, while Sect. 8 presents results for max-cut. Finally, concluding remarks follow in Sect. 9.

2 Background

Consider a solution s of a combinatorial optimization problem. Let $f(s)$ be the cost of solution s . Let N be a neighborhood operator that from a given solution produces a set of modified solutions. Thus, the neighborhood of s is $N(s)$. A pseudo-code of a local search procedure for a minimization problem is presented in Algorithm 2.1. A solution returned by the local search is a local optimum of the considered neighborhood.

Algorithm 2.1: Local search.

```
1 Local Search( $s$ ) {  
2   while there is  $s' \in N(s)$  such that  $f(s') < f(s)$  do  
3      $s \leftarrow s'$ ;  
4   return  $s$ ;  
5 }
```

Many improving solutions may be available in $N(s)$ at a given iteration of the search. The best-known strategies for local search differ in the selection of the improving neighbor, which often guides the local search to local optima of different quality.

The BILS selects at each iteration the neighbor with the best objective function value. With this strategy the number of iterations of the local search tends to be smaller than with other approaches, as the objective function value of the current solution is improved as much as possible in each iteration. On the other hand, it requires the evaluation of the whole neighborhood.

The FILS selects at each iteration the first evaluated neighbor that is better than the current solution. In this case, the order in which the neighborhood is evaluated influences the search. Observe that if the neighborhood is evaluated in a random order, first improvement is equivalent to a random selection of an improving neighbor.

One of these two strategies is implemented in the vast majority of the local search and metaheuristics literature. Hansen and Mladenović (2006) analyzed and computationally compared these two strategies along with worst improvement (selection of the least improving neighbor) in the context of 2-opt for the TSP. They concluded that FILS is the best method when the search starts from a random initial solution, whereas BILS is better when the initial solution is constructed with a greedy heuristic.

In the context of local search strategies it is important to mention the variable neighborhood descent (VND) (Duarte et al. 2018) procedure. VND defines a strategy for performing local search on the union of several neighborhoods. Since the search stops only when a local optimum of all the considered neighborhoods is found, the quality of the final solution is in general better than the one obtained with a local search using a single neighborhood.

Several techniques have been introduced over the years to improve the effectiveness of local search and prevent it from getting stuck in low-quality local optima. These include metaheuristics such as simulated annealing (Kirkpatrick et al. 1983; Černý 1985), guided local search (Voudouris 1997), tabu search (Glover 1986, 1989, 1990),

variable neighborhood search (Mladenović and Hansen 1997), and iterated local search (Lourenço et al. 2010).

When used in the context of a metaheuristic, not only the quality of the returned solution is important but also the speed of execution of the search as a whole. The FILS focuses on the speed by evaluating as few neighbors of the current neighborhood as possible. On the other hand, BILS focuses on decreasing the number of iterations of the search by improving the objective function as much as possible. The literature discusses several techniques to speed-up a neighborhood evaluation, such as delta evaluations (the evaluation of a neighbor based on the difference in cost compared to the current solution) (Bertsimas 1988), heuristic evaluations (an approximate evaluation of the cost function instead of its exact evaluation) (Prandstetter and Raidl 2008; Solnon et al. 2008), candidate lists (evaluating first the neighbors that are considered most likely to be improving) (Pekny and Miller 1994), and neighborhood reductions (a priori excluding neighbors that can be proven to be non-improving) (Nowicki and Smutnicki 1996).

Both BILS and FILS can be seen as greedy, selecting the most improving neighbor in the case of BILS and the first available in FILS. Those greedy decisions can lead the search to a local optimum of bad quality. The approach to be presented in the next section decides in a less greedy way on which neighbor to select as the next current solution at each iteration. It moves the search to an improving neighbor that has the greatest potential for further improvements when applying the same neighborhood operator for making subsequent moves. The new approach relies on the availability of fast to execute local optimality tests. Lancia et al. (2015) introduced a set of tests for the 2-opt neighborhood of the TSP and for the 1-flip neighborhood of max-cut.

It is important to remark that the approach being proposed is not a new metaheuristic. Metaheuristics based on local search implement mechanics to escape from local optima. Our approach, as any other local search procedure, stops as soon as a local optimum of the considered neighborhood becomes the current solution.

3 Delayed improvement strategy

Define an attribute $a(s)$ to be a function that retrieves a specific part of a solution s . Let a *local optimality precondition check* $p(a(s))$ be a predicate on a given attribute of a feasible solution such that if s is a local optimum of a given neighborhood N then, for any attribute function a , $p(a(s))$ evaluates to true. In other words, p tests a condition that is always satisfied for local optima of N .

Consider a set $A = \{a_1, a_2, \dots, a_m\}$ of attributes and let $P = \{p_1, p_2, \dots, p_t\}$ be a set of local optimality precondition checks of N . Define $A_i \subseteq A$ as the subset of attributes that are relevant for predicate p_i . Let $k_P(s) = |\{(i, j) : p_i(a_j(s)) = \text{false}, 1 \leq i \leq t, a_j \in A_i\}|$ be the number of checks in P that evaluate to false for a solution s . By definition, $k_P(s) = 0$ for every local optimum s of neighborhood N .

The DILS selects at each iteration, a solution s' that maximizes $k(s')$ among all improving solutions of $N(s)$. Solution cost is used as tie-breaker. Algorithm 3.1 depicts the proposed approach.

Algorithm 3.1: Delayed improvement local search.

```

1 Delayed Improvement Local Search(s)
2 while  $I(s) = \{s' \in N(s) : f(s') < f(s)\}$  is not empty do
3    $k \leftarrow \max_{s' \in I(s)} k_P(s')$ ;
4    $s'' \leftarrow \operatorname{argmin}_{s' \in I(s), k_P(s')=k} f(s')$ ;
5    $s \leftarrow s''$ ;
6 return s;

```

The idea behind the method is to improve the current solution while avoiding, as much as possible, the convergence to a low-quality local optimum. That is, the search should be given more opportunities of finding high quality solutions by slaloming local optima of bad quality. It may seem counter-intuitive to avoid local optima, since the final objective of DILS, as with any local search, is to find a locally optimal solution. However, local optima that can be found within a few improving moves from the initial solution may not be as good as those that can be obtained after making many improving steps. DILS aims to locate a local optimum that is far from the initial solution and that is obtained only after making many improving moves. This is achieved by improving the solution while remaining as far away as possible from other local optima that would halt the search.

The approach has a potential drawback in terms of running time. Since all checks need to be evaluated for all candidate solutions, each iteration of the search is computationally more demanding than in BILS. However, this effect can be diminished by performing the evaluation of the checks only after the cost evaluation of the candidate solution and only if the candidate solution is improving. Since local optima are being avoided, the expected number of iterations required to reach a local optimum tends to be higher in DILS than in BILS or FILS. Thus, to be computationally effective, the proposed local search approach needs to attain on average better results than the classic approaches. In this paper we analyse the computational performance of DILS in implementations of the 2-opt neighborhood for the TSP and the 1-flip neighborhood for the max-cut problem.

Both BILS and FILS select at each iteration the evaluated solution with the best cost. After any iteration the current solution is the best ever evaluated since the start of the search. This property does not hold for DILS and, in consequence, it may happen that the returned solution at the end of the search is not the best ever evaluated: one of the evaluated neighbors in a previous step may have been better than the final local optimum. While this behavior is theoretically possible, it is extremely rare in practice, and it is therefore not worth spending computational resources on storing any additional solutions encountered during the local search.

4 Delayed improvement applied to the traveling salesman problem

The TSP is among the most studied combinatorial optimization problems. Given a set V of points and the (symmetric) distances between them, the problem consists

in obtaining a tour of minimum length through all the points in V . It is an NP-hard problem and, in consequence, no polynomial-time exact algorithm is expected to exist. The TSP and its variations are very relevant in practice and their solution is part of many real-world operations research applications. For convenience, we define the TSP on a complete undirected graph G in which the vertices represent the points of the instance. Then, a solution to the TSP is a Hamiltonian cycle of G .

Many algorithms, both exact and heuristic, have been proposed for the TSP (Laporte 1992; Kumar and Panneerselvam 2012; Rego et al. 2011). Among those algorithms there are several constructive heuristics and local search heuristics. For the latter, the 2-opt neighborhood is commonly used (Johnson and McGeoch 1997)

The 2-opt neighborhood consists of all the solutions that can be obtained from s by eliminating two non-consecutive edges of s and reconnecting the two obtained paths to form a new tour. Observe that a solution s and a neighbor of s differ in that one part of the tour (the one in between the two eliminated edges) is inverted in relation to each other. Thus, the cost difference of a solution and a neighbor from the 2-opt neighborhood is given by the cost of just four edges, the two eliminated from the original solution and the two inserted to reconnect the tour.

To apply the DILS strategy to the 2-opt search we need, as introduced in the previous section, a set of local optimality precondition checkers. Lancia et al. (2015) introduced local search inequalities in the context of the 2-opt neighborhood for the TSP. These inequalities are true statements for all local optima relative to the 2-opt neighborhood. The authors used the inequalities to improve the performance of a branch and cut algorithm. Including the inequalities in an integer linear programming model does not cut away any 2-opt local optimum and, in consequence, despite cutting feasible solutions from the search space, no optimal solution is excluded. In this work, we propose to use a subset of the inequalities presented by Lancia et al. (2015) as local optimality precondition checkers.

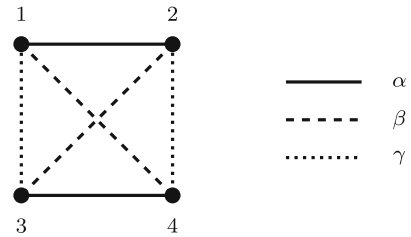
4.1 Local optimality precondition checkers

We now describe the inequalities proposed by Lancia et al. (2015) following their notation. Given any subset S of V with four vertices (without loss of generality assume $S = \{1, 2, 3, 4\}$), we split the possible edges among the points in S in three disjoint pairs of edges: the horizontal edges α , the crossing edges β and the vertical edges γ (see Fig. 1). Define $c(p)$ as the sum of the cost of each edge in the pair p , assuming, without loss of generality, that $c(\alpha) \geq c(\beta) \geq c(\gamma)$. Given a solution s we denote respectively by $x(\alpha)$, $x(\beta)$ and $x(\gamma)$ the number of edges of α , β and γ present in s . Note that $x(\alpha) + x(\beta) + x(\gamma) \leq 3$ for any solution s whenever the total number of points in the instance is greater than four.

Proposition 1 in Lancia et al. (2015) states that if $c(\alpha) > c(\beta)$, then inequalities $x(\alpha) \leq 1$ hold for any 2-opt locally optimal solution s . Then, we can create a local optimality precondition checker p_1 that takes as input any four vertices of the graph and checks whether $(c(\alpha) > c(\beta)) \rightarrow (x(\alpha) \leq 1)$.

The second proposition from Lancia et al. (2015) states that if $c(\beta) > c(\gamma)$, then $x(\alpha) + x(\beta) \leq 2$ is valid for any 2-opt locally optimal solution s . Then, we create a

Fig. 1 Subsets α , β and γ of $E(1, 2, 3, 4)$



local optimality precondition checker p_2 that, as p_1 , takes as input any four vertices of the graph and checks whether $(c(\beta) > c(\gamma)) \rightarrow (x(\alpha) + x(\beta) \leq 2)$ holds.

Proposition 3 from Lancia et al. (2015) states a set of three types of inequalities that hold for 2-opt locally optimal solutions whenever $c(\alpha) > c(\beta) > c(\gamma)$. From each of these types of inequalities we create a number of new local optimality precondition checkers as follows.

Let $x(e)$ be equal to 1 if edge e is in solution s and equal to 0 otherwise. From the inequalities $2x(\alpha) + x(\beta) + x(e_\gamma) \leq 3$ for each $e_\gamma \in \gamma$, we create two checkers. Local optimality precondition checker p_{3a} checks $(c(\alpha) > c(\beta) > c(\gamma)) \rightarrow (2x(\alpha) + x(\beta) + x(\gamma_1) \leq 3)$ and checker p_{3b} checks $(c(\alpha) > c(\beta) > c(\gamma)) \rightarrow (2x(\alpha) + x(\beta) + x(\gamma_2) \leq 3)$ where γ_1 and γ_2 are the edges of γ taken in lexicographical order.

From the inequalities $2x(\alpha) + 2x(e_{\beta_1}) + x(e_{\beta_2}) + x(\gamma) \leq 4$ for each $e_{\beta_1}, e_{\beta_2} \in \beta, e_{\beta_1} \neq e_{\beta_2}$, we create two checkers. Local optimality precondition checker p_{4a} checks $(c(\alpha) > c(\beta) > c(\gamma)) \rightarrow (2x(\alpha) + 2x(\beta_1) + x(\beta_2) + x(\gamma) \leq 4)$ and checker p_{4b} checks $(c(\alpha) > c(\beta) > c(\gamma)) \rightarrow (2x(\alpha) + 2x(\beta_2) + x(\beta_1) + x(\gamma) \leq 4)$ where β_1 and β_2 are the edges of β taken in lexicographical order.

Finally, from inequalities $3x(\alpha) + 2x(\beta) + x(\gamma) \leq 5$ we create the local optimality precondition checker p_5 that checks $(c(\alpha) > c(\beta) > c(\gamma)) \rightarrow (3x(\alpha) + 2x(\beta) + x(\gamma) \leq 5)$. In Lancia et al. (2015) a fourth set of local search inequalities is also proposed. However, the number of inequalities of that type grows exponentially with the instance size, and we opt not to use them due to the amount of computational work associated with their verification.

Algorithm 4.1 depicts the application of DILS for TSP. The algorithm takes an initial solution and iterates until no improving neighboring solution is found, just like any local search algorithm. At each iteration, for each improving neighbor the algorithm computes the number of local optimality precondition checkers that are not satisfied. The improving neighbor with maximum number of violated local optimality precondition checkers is selected as the next current solution. Ties are broken by the best value for the objective function.

4.2 Implementation details

The main drawback of DILS is the computational effort associated with the evaluation of the local optimality precondition checkers for every neighboring solution. A naive implementation of such an evaluation would produce an $O(n^4)$ procedure to compute

Algorithm 4.1: Delayed improvement local search for TSP.

```

1 Delayed Improvement Local Search for TSP(s)
2 while  $I(s) = \{s' \in N(s) : f(s') < f(s)\}$  is not empty do
3    $k \leftarrow -1$ ;
4    $c \leftarrow \infty$ ;
5   foreach  $s' \in I(s)$  do
6      $k' \leftarrow 0$ ;
7     foreach all distinct  $a, b, c, d \in V$  do
8        $k' \leftarrow k' + (p_1(a, b, c, d) = \text{false})$ ;
9        $k' \leftarrow k' + (p_2(a, b, c, d) = \text{false})$ ;
10       $k' \leftarrow k' + (p_{3a}(a, b, c, d) = \text{false})$ ;
11       $k' \leftarrow k' + (p_{3b}(a, b, c, d) = \text{false})$ ;
12       $k' \leftarrow k' + (p_{4a}(a, b, c, d) = \text{false})$ ;
13       $k' \leftarrow k' + (p_{4b}(a, b, c, d) = \text{false})$ ;
14       $k' \leftarrow k' + (p_5(a, b, c, d) = \text{false})$ ;
15     if  $k' > k$  or ( $k' = k$  and  $f(s') < c$ ) then
16        $s'' \leftarrow s'$ ;
17        $c \leftarrow f(s')$ ;
18        $k \leftarrow k'$ ;
19    $s \leftarrow s''$ ;
20 return s;

```

the number of local optimality precondition checkers that evaluate to false for each neighbor.

However, inspecting the checkers, one can verify that all of them, in order to be evaluated to false, need at least two non-consecutive edges involving nodes a , b , c , and d . Then, one can iterate over all the pairs of edges of the solution when evaluating the checkers. Thus, the computation time required for the evaluation of all the checkers is reduced to $O(n^2)$.

Moreover, in the context of a local search, as with the objective function, after the evaluation of the first solution, there is no need to evaluate the exact number of false checkers but only the difference between the number of false checkers of the current solution and of the neighboring solution being evaluated. Let us call this number Δ . In the case of 2-opt, just two edges are exchanged from the current solution to each neighbor. Then, to compute Δ , one needs to evaluate only checkers for nodes a , b , c , and d that are the endpoints of edges that were added or removed from the current solution.

Furthermore, in the context of a 2-opt local search, all checkers can be evaluated for a single neighbor in $O(n)$ steps, since we just consider all pairs of non-consecutive edges that are present in one solution and not in the other one. As only two edges are exchanged, the numbers of pairs to be considered is limited by $4n$.

In this work, we improve further upon this, by evaluating Δ for each neighboring solution in amortised constant time, yielding $O(n^2)$ time to evaluate all the $O(n^2)$ solutions in the neighborhood. To achieve this, we compute and store two values for each pair of non-consecutive edges in the solution in each iteration. The first value is the Δ associated with the corresponding 2-opt move. The second one, Δ' , is the number of false checkers obtained when performing an invalid reconnection of the

paths after the pair of edges is removed. In this case, we are saving the number of checkers evaluated to false for infeasible solutions that are composed of two cycles.

With that information stored we can now select the improving move with larger Δ , breaking ties with the objective function value. In the next iteration the stored values will be used to update the values of Δ and Δ' of each move.

The new current solution is very similar to the previous one. It differs in the exchange of a pair of edges. Let e and f be the edges that were present in the previous solution and are not present in the current solution. Also, let g and h be the edges that replaced e and f to obtain the new solution. Consider first the updating of Δ and Δ' for a pair of non-consecutive edges (say i and j) both different from the new edges g and h . The values of Δ and Δ' for that pair may indeed change, as one must compensate for the number of checkers that would be affected before the most recent move but not after, and vice versa.

This means that additional calculations, to update Δ and Δ' , must be performed for combinations of edges involving one of the edges i or j , as well as one of the four edges g , h , e , or f . Since there are four edges involved in the evaluated move (i , j and the edges that would replace them) and four edges that went in or out of the solution in the last iteration, there is a total of 16 pairs of edges to be considered when updating Δ , as well as 16 pairs of edges to be considered for updating Δ' . The number of pairs is constant, and therefore the cost of updating Δ and Δ' is also constant.

Depending on the update of the solution in the last iteration, the move involving i and j may now need to add different edges than before in order to reconnect the two paths that appear as a result of removing them. In those cases one must use Δ' as the base to update Δ and Δ as the base to update Δ' .

It remains to compute Δ and Δ' for the moves involving the new edges g and h introduced in the solution in the last iteration. We do not have any previous information about Δ and Δ' for these moves so we must compute each value of Δ and Δ' from scratch in $O(n)$ time. Given that there are $O(n)$ moves involving the two new edges the total time for evaluating those moves is $O(n^2)$.

Thus, we have $O(n^2)$ moves for which we can evaluate Δ and Δ' in constant time and $O(n)$ moves that can be evaluated in $O(n)$. The full evaluation of the neighborhood is then $O(n^2)$ which gives us $O(1)$ amortised time for each move.

5 Delayed improvement applied to the max-cut problem

The cardinality max-cut problem is the following: Given a graph $G = (V, E)$, split V into sets S and \bar{S} such that the number of edges with one end in S and the other in \bar{S} is maximized. The problem is strongly NP-Hard (Garey et al. 1976). Heuristics, approximation algorithms, and exact approaches have been proposed for max-cut. Most of the literature considers the weighted version of the problem, in which the sum of the weights of the edges with endpoints in both sets has to be maximized, while we consider the unweighted problem. A local search heuristic using the 1-flip neighborhood was proposed by Festa et al. (2002).

The 1-flip neighborhood consists of all the solutions that can be obtained from a solution s by moving one vertex from S to \bar{S} or from \bar{S} to S . The cost difference of

a solution and a neighbor obtained by moving a vertex v from one set to the other, is given by the difference between the number of vertices adjacent to v that are in the same set as v and the number of vertices adjacent to v that are in the other set.

5.1 Local optimality precondition checkers

To apply the DILS strategy to the 1-flip search we use a local optimality precondition checker derived from the local search inequality introduced by Lancia et al. (2015): in a local optimum for the 1-flip neighborhood, no vertex can have more adjacent vertices in its own set than in the other set, otherwise moving the vertex to the other set would improve the solution.

Let $\delta(v)$ be the vertices adjacent to v in G and $T(s, v)$ be the vertices of G that are in the same set as v in solution s . Then, the local optimality precondition checker p checks, for a given $v \in V$, whether $|\{u : u \in \delta(v), u \in T(s, v)\}| \leq |\delta(v)|/2$.

Algorithm 5.1 describes the application of DILS to max-cut. The algorithm takes an initial solution and iterates until no improving neighbor is found. At each iteration, for each improving neighbor the algorithm computes the number of local optimality precondition checkers that are not satisfied. The improving neighbor with the maximum number of violated local optimality precondition checkers is selected as the next solution to visit. Ties are broken by the best value for the objective function.

Algorithm 5.1: Delayed improvement local search for max-cut.

```

1 Delayed Improvement Local Search for max-cut( $s$ )
2 while  $I(s) = \{s' \in N(s) : f(s') < f(s)\}$  is not empty do
3    $k \leftarrow -1$ ;
4    $c \leftarrow \infty$ ;
5   foreach  $s' \in I(s)$  do
6      $k' \leftarrow 0$ ;
7     foreach  $v \in V$  do
8        $k' \leftarrow k' + (p(v) = \text{false})$ ;
9     if  $k' > k$  or ( $k' = k$  and  $f(s') < c$ ) then
10       $s'' \leftarrow s'$ ;
11       $c \leftarrow f(s')$ ;
12       $k \leftarrow k'$ ;
13    $s \leftarrow s''$ ;
14 return  $s$ ;
```

5.2 Implementation details

In a naive implementation of BILS or FILS using the 1-flip neighborhood, one would just store, for each vertex, the difference between the number of adjacent vertices in the same set and the number of adjacent vertices in the other set. That difference,

which we call $\Delta(v)$ (or just Δ when v is obvious), is the improvement of the objective function value if v is moved to the other set.

That implementation can be improved by computing and updating a list of vertices for which Δ is larger than 0. All improving neighbors can be obtained by moving a vertex from this list. When working with relatively good solutions, such as those produced by a greedy construction heuristic, the number of vertices with a positive Δ is much smaller than the number of vertices in the graph. Then, FILS can simply choose the first vertex on the list, while BILS can look for the vertex on the list with the maximum value of Δ .

One may think that a better way to implement BILS is by maintaining a sorted list of vertices with $\Delta > 0$. However, when working with graphs of density around 0.5, corresponding to the most interesting instances for cardinality max-cut, the values of Δ changes for about half of the vertices after every move. It is therefore cheaper to scan an entire unsorted list each iteration, rather than updating a sorted list after every iteration.

To implement DILS we need to compute, besides Δ , the number of vertices that would have $\Delta > 0$ if vertex v is moved. Observe that this number is exactly the number of local optimality precondition checkers that the neighbor does not satisfy. Instead of computing this number directly, we compute the difference in unsatisfied checkers of each improving neighboring solution in relation to the current solution, which we refer to as Δ' .

It is more difficult to update Δ' than to update Δ . After each move, Δ is only updated for the vertex being moved and its adjacent vertices. However, Δ' may change also for the vertices adjacent to the vertices adjacent to the moved vertex. In high density graphs the update operations would then involve almost all the vertices of the graph. However, Δ' is only relevant for improving neighbors, i.e., the neighbors obtained by moving a vertex with a positive Δ . Then, our proposal is to compute Δ' just for the vertices with a positive Δ .

For computing Δ' we use an additional list of vertices that have $\Delta \in \{-1, 0, 1, 2\}$. These vertices are the ones that may go from a positive to a non-positive Δ or from a non-positive to a positive Δ after the move of an adjacent vertex and, in consequence, need to be considered in the computation of the Δ' value of its adjacent vertices.

6 Experimental setup

To analyze DILS we design two types of experiments, attempting to highlight either the behavior of the method or the performance of the method. These experiments are both conducted on instances of the TSP and the max-cut, and the DILS is compared to the corresponding behavior or performance of BILS and FILS.

To investigate the behavior of DILS we execute DILS, BILS, and FILS on a single test instance starting from the same initial solution. In these tests we compute the number of checkers not satisfied at each iteration for all of the three methods. This enables us to show how the three methods differ in terms of their progressions of solution quality and the number of violated checkers at each iteration of the search. Furthermore,

to investigate the behavior, we analyze the number of iterations performed to reach a local optimum by each of the methods.

When analyzing the performance of DILS, we consider the local search methods in a multi-start framework. Each method is restarted when having reached a local optimum, while recording the value of the best local optimum encountered. The time consumption of DILS may be higher than the time consumption for the other methods due to having a higher computational effort per iteration while trying to execute more iterations before converging to a local optimum. Therefore, when evaluating the performance of DILS, it is unfair to run each of the methods for any fixed number of restarts.

Instead, we consider two different settings for making fair comparisons. In the first setting we fix a running time limit and repeatedly execute a given local search (BILS, FILS, or DILS) from different initial solutions until the time limit is reached. The best solution found is returned. This multi-start approach is a fairer comparison of the methods, since the faster methods can be repeated more times than the slower ones, thereby increasing their probability of finding a good local optimum.

In the second setting we consider individual instances and produce time to target (TTT) plots (Aiex et al. 2007). To create these plots we set a target solution cost and measure the time needed (up to a high overall time limit) for each method to reach a solution with an objective function value at least as good as the target. We repeat the experiment a large number of times and sort the time to target values in ascending order. Plotting these data we get an estimate of the probability $p(t)$ of the algorithm obtaining a solution with the desired quality as a function of the running time.

6.1 Test instances

For both the TSP and the max-cut we consider 30 test instances. In the case of TSP they come from the TSPLib (Reinelt 1991). For max-cut the instances are randomly generated and consist of graphs with the number of nodes ($|V|$) ranging from 1500 to 2500 and a density (D) ranging from 0.45 to 0.54.

6.2 Construction heuristics

The local search procedures start from an initial solution. In this work we use two different methods to obtain these initial solutions for TSP. The main method is an insertion heuristic. It starts with a tour consisting of a randomly selected vertex and the vertex that is nearest to it. Then, during $n - 2$ iterations, a random vertex not yet in the tour is inserted in the position which increases the length of the tour the least. An alternative approach constructs an initial solution by creating a random permutation of the vertices. In general, the main method produces much better solutions than the random permutation.

For max-cut we use a construction heuristic that randomly permutes the nodes of the graph and, in the obtained order, adds the next node to the best of the two sets while considering only the nodes previously included in the solution.

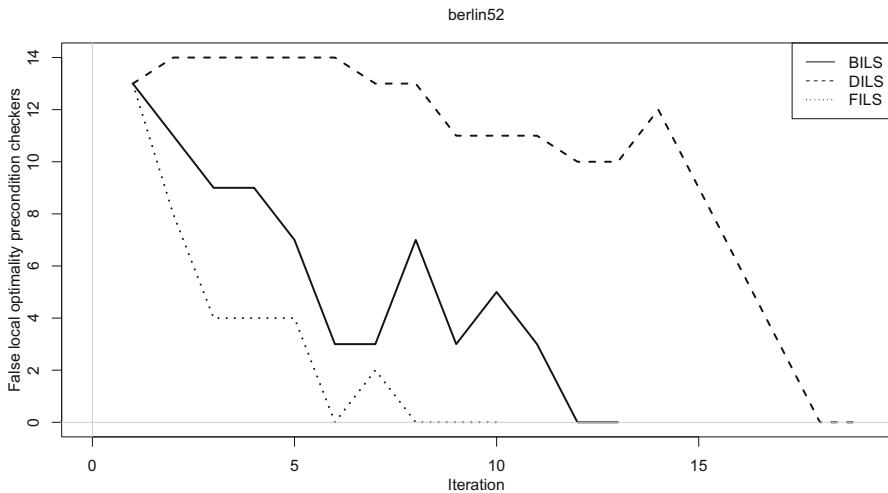


Fig. 2 Number of checkers not satisfied for the berlin52 instance, with the initial solution created by a constructive heuristic

6.3 Computational setup

The local search methods were implemented using the C programming language. The code was compiled with the gcc compiler using the O3 optimization flag. The experiments were run on a 64bits Intel(R) Xeon(R) E5405, 2GHz machine with 4 processors and 16.0GB of RAM running the Ubuntu 18.04.3 LTS operating system. The methods were all programmed according to the same programming standards and with the same level of optimization.

7 Computational results for TSP

Computational results for the TSP are based on two types of experiments. The tests presented in Sect. 7.1 are designed to show the behavior of DILS, while the tests in Sect. 7.2 compare the performance of DILS with the performance of BILS and FILS.

7.1 Search behavior

We first consider the behavior of DILS on two illustrative test instances. Figures 2 and 3 show the number of checkers not satisfied by the current solution at each iteration in a single execution for instances berlin52 and pr299, respectively. The initial solution was obtained with the constructive heuristic. Figures 4 and 5 show, for the same executions of each search, the objective function value of the current solution at each iteration of the search.

As expected, the number of iterations is larger for DILS than for the other methods. The number of violated checkers goes up during the first few iterations of DILS,

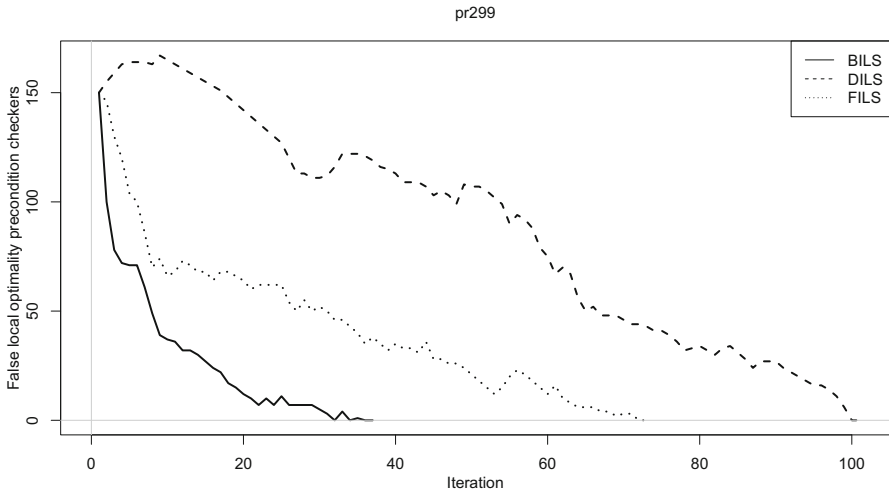


Fig. 3 Number of checkers not satisfied by the current solution for the pr299 instance, with the initial solution created by a constructive heuristic

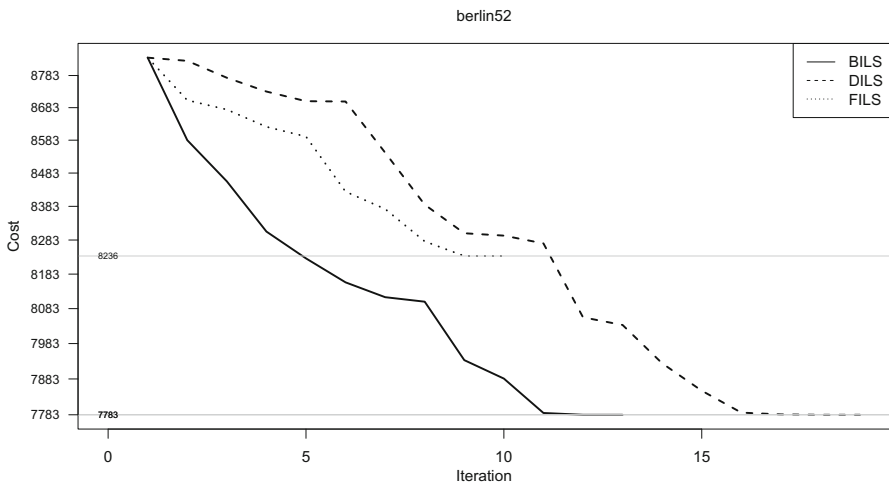


Fig. 4 Objective function value of the current solution for the berlin52 instance, with the initial solution created by a constructive heuristic

meaning that the search is able to find improving solutions with fewer properties associated to local optima. At some point, this is not possible anymore and the search slowly approaches a local optimum. In Figs. 4 and 5 it can be seen that, even though the improvement in the objective function is slower than for the other methods, the longer search performed by DILS can lead to finding local optima with the same or better quality than the ones otherwise obtained.

In Fig. 6 we show the same information as in Fig. 3, for the same instance but with the search starting from a random solution. We can observe that the number of

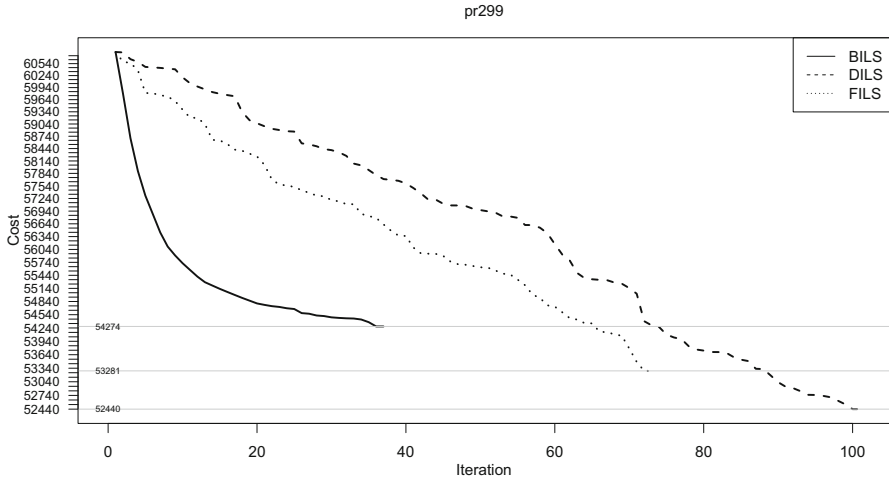


Fig. 5 Objective function value of the current solution for the pr299 instance, with the initial solution created by a constructive heuristic

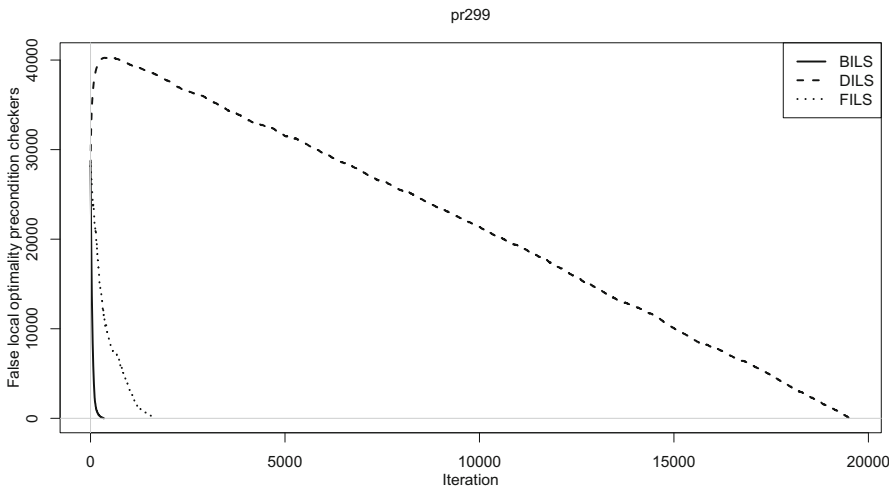


Fig. 6 Number of checkers not satisfied by the current solution for the pr299 instance, with the initial solution randomly generated

iterations performed by DILS is much larger than the the number of iterations for the other methods. In the particular case plotted in the figure, DILS performs around 100 times more iterations than BILS. This ratio becomes even larger when the instances increase in size.

A random solution has, in general, a larger number of improving neighbors than a solution obtained by a constructive heuristic. Thus, in the first iterations, DILS is able to find improving neighbors that increase the number of checkers not satisfied, after which it very slowly convergences to a local optimum. The large differences

Table 1 Number of iterations performed by each search

	Inst.	BILS	FILS	DILS	DILS/BILS
1	berlin52	6.41	8.96	14.02	2.19
2	ch130	13.28	19.43	27.86	2.10
3	ch150	14.15	21.42	30.4	2.15
4	d1291	75.74	104.23	134.67	1.78
5	kroA200	20.3	29.95	41.01	2.02
6	kroB200	18.56	27.63	37.46	2.02
7	kroC100	11.8	17.25	24.01	2.03
8	kroD100	11.2	16.13	23.11	2.06
9	lin105	10.85	16.37	25.13	2.32
10	linhp318	24.83	36.24	52.69	2.12
11	p654	42.42	69.44	113.41	2.67
12	pcb1173	83.06	127.64	172.59	2.08
13	pcb442	27.43	44.62	63.14	2.30
14	pr1002	80.49	114.38	152.87	1.90
15	pr152	13.15	18.79	28.18	2.14
16	pr2392	163.92	236.39	316.95	1.93
17	pr299	25.62	39.7	54.75	2.14
18	pr439	37.41	58.48	83.31	2.23
19	rat195	15.19	23.25	31.3	2.06
20	rd400	32.41	47.36	62.1	1.92
21	rl1323	87.93	135.67	195.34	2.22
22	rl1889	118.78	179.46	243.34	2.05
23	tsp225	18.08	27.65	38.26	2.12
24	u1060	78.54	114.14	161.68	2.06
25	u1432	47.96	70.25	90.22	1.88
26	u1817	94.74	140.91	179.21	1.89
27	u2152	102.28	146.33	195.97	1.92
28	u2319	40.96	64.29	80.77	1.97
29	u574	47.86	68.29	93.66	1.96
30	vm1748	96.59	140.71	188.94	1.95

Average of 100 restarts

in the number of iterations performed before a local optimum is reached, plus the extra cost for evaluating the checkers, makes it improbable that DILS can compensate for the difference in computational time by the quality of the solution obtained. In consequence, we conclude that DILS is not competitive when the initial solution is randomly generated, and the rest of the experiments consider initial solutions obtained with the construction heuristics.

Table 1 gives the average number of iterations for each method over 100 restarts. The number of iterations of DILS is close to twice the number of iterations of BILS in all the instances considered, with an average ratio of 2.1.

7.2 Search performance

In the first set of experiments we fix a running time limit of 300 s and execute the local search methods in a multi-start framework. To get more reliable results, each instance is solved ten times by each of DILS, BILS, and FILS. As observed, DILS requires more iterations to reach a local optimum. In the experiments with a fixed time limit, the number of restarts executed by DILS is an order of magnitude smaller than the number of restarts by BILS and FILS.

In Table 2 we report both the average cost of the solutions found over the ten executions (Avg. Z) and the cost of the best solution in the ten executions (Min. Z). When we consider the minimum cost over the ten runs for each instance, DILS obtained the best results for 25 instances, while the BILS and the FILS found the best solutions for only 10 instances each. By applying a chi-squared test, the hypothesis that the performance of DILS is equal to the performance of BILS is rejected, with a P value of 6×10^{-9} . The same conclusion holds when comparing DILS to FILS. When considering average costs over the 10 executions, the number of best performances is 17 for DILS, 12 for BILS, and 9 for FILS. Applying again a chi-squared test, the hypothesis that the average performance of DILS is equal to the average performance of FILS is rejected with a P value of 0.001, whereas the same test fails to reject the hypothesis that DILS and BILS could have the same average performance as the P value is 0.06.

For the runs in Table 2 we also calculate the percentage gap to the best solution found for each instance within the experiment. The table then reports both the average and the standard deviation of these gaps across the 30 instances, considering both the average and the best objective function values found within the ten executions. We can then apply Welch's t -test for the hypothesis that the gaps are equal for any pair of methods. These tests show that the average gaps are not significantly different for any pair of methods, but the best out of ten executions is better for DILS than the other two methods, with P values less than 0.005. In conclusion, DILS overall performs better than the other methods on the tested instances.

The second set of performance evaluation tests considers time to target (TTT) plots (Aiex et al. 2007). We use an overall time limit of 900 s, and perform 100 runs that end by either reaching the chosen target or by reaching the time limit. Figures 7 to 9 show TTT plots for instances rat195, tsp225 and rd400. Four plots are shown for each instance considering progressively harder targets. The hardest target is set to the best average solution cost over the three algorithms in Table 2 rounded to the nearest integer. The other three targets are 1%, 2% and 3% above the hardest target, rounded to the nearest integer.

The plots show that when the target is easy, BILS and FILS perform better than DILS. However, all the three algorithms reach the target in a few seconds. On the other hand, when the target gets harder to achieve, the performance of DILS improves relative to BILS and FILS. With the hardest target, the superior performance of DILS is clear. We can conclude that while DILS is slower than the classic strategies to obtain average quality solutions, in the long run it tends to achieve solutions of better quality.

Table 2 Results for DILS, BILS and FILS

	Instance	BILS		FILS		DILS	
		Avg. Z	Min. Z	Avg. Z	Min. Z	Avg. Z	Min. Z
1	berlin52	7542.0	7542	7542.0	7542	7542.0	7542
2	ch130	6117.6	6110	6117.0	6110	6120.4	6110
3	ch150	6577.3	6561	6566.0	6548	6574.9	6548
4	d1291	55200.8	54935	55401.4	54961	55287.6	54901
5	kroA200	29678.0	29573	29686.2	29599	29683.3	29610
6	kroB200	29728.2	29645	29733.3	29632	29793.0	29632
7	kroC100	20749.0	20749	20749.0	20749	20749.0	20749
8	kroD100	21294.0	21294	21296.8	21294	21308.4	21294
9	lin105	14379.0	14379	14379.0	14379	14379.0	14379
10	linhp318	43245.2	43103	43255.2	43059	43245.9	42958
11	p654	35049.4	35007	35069.7	35049	35069.4	34915
12	pcb1173	62440.5	61885	62175.9	61866	61765.1	61141
13	pcb442	53506.1	53383	53495.1	53117	53038.3	52715
14	pr1002	275906.2	274720	276056.4	274975	275679.4	274353
15	pr152	73682.0	73682	73682.0	73682	73682.0	73682
16	pr2392	416929.7	414049	415425.0	412988	415618.1	413393
17	pr299	49185.2	48943	49179.5	49011	49220.8	49117
18	pr439	110308.6	109730	110489.4	109937	110048.8	109394
19	rat195	2405.2	2387	2402.2	2380	2399.8	2389
20	rd400	15908.6	15856	15904.4	15835	15875.7	15811
21	rl1323	290386.3	285853	291240.3	287928	289982.0	285853
22	rl1889	346631.9	343891	348353.9	346499	344231.4	339537
23	tsp225	4036.9	4031	4041.2	4022	4037.5	4020
24	u1060	238396.8	237519	238504.0	237602	237519.0	236893
25	u1432	166572.3	165792	166646.2	166054	166621.3	166282
26	u1817	64391.0	64083	64514.0	64137	64380.7	63945
27	u2152	73100.0	72417	73213.5	72675	72780.1	72361
28	u2319	248396.3	247856	248218.4	247735	248246.1	247533
29	u574	38865.3	38745	38821.6	38597	38807.8	38548
30	vm1748	363912.0	361293	363577.3	362488	362627.3	358457
	No. of best	12	10	9	10	17	25
	Avg. % gap	0.69	0.27	0.71	0.30	0.53	0.04
	Std. % gap	0.58	0.37	0.61	0.45	0.37	0.10

Performing for 300 s, 10 repetitions, starting from solution generated from the constructive heuristic. The best obtained solution for each instance is highlighted in bold

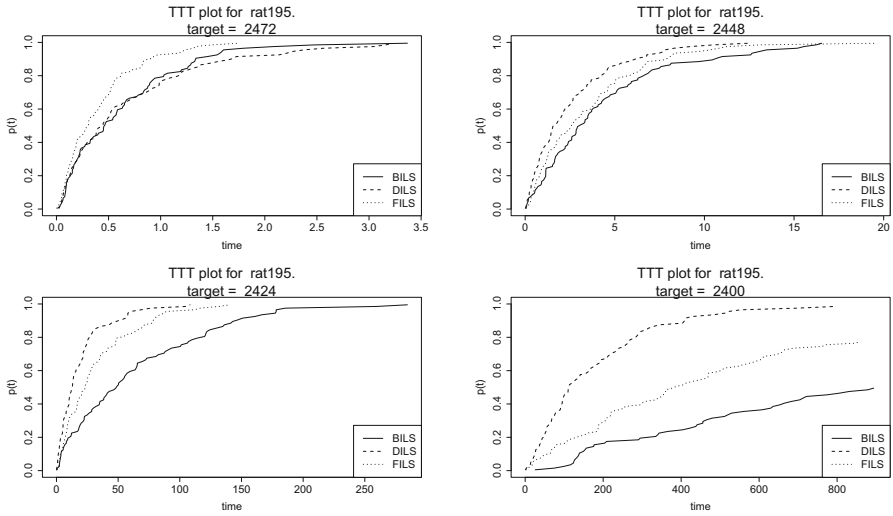


Fig. 7 TTT plots for rat195, with easiest target on the top left and the hardest target on the bottom right

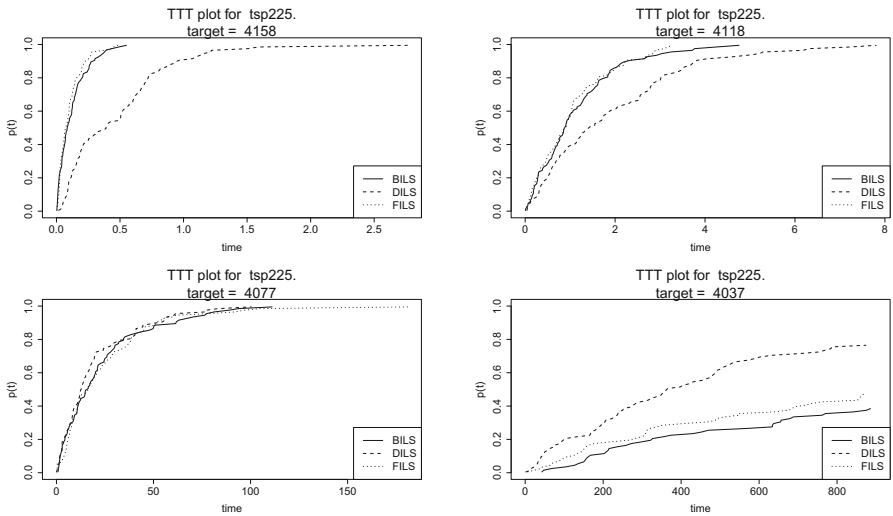


Fig. 8 TTT plots for tsp225, with easiest target on the top left and the hardest target on the bottom right

8 Computational results for max-cut problem

Computational results for the max-cut problem are presented based on the same two types of tests as for the TSP.

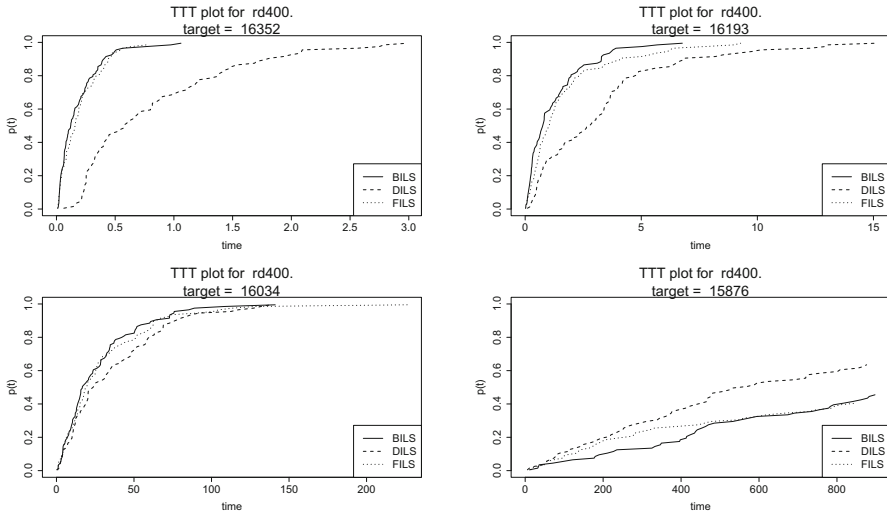


Fig. 9 TTT plots for rd400, with easiest target on the top left and the hardest target on the bottom right

8.1 Search behavior

To investigate the behavior of DILS we performed the same type of tests as for TSP. We executed DILS, BILS and FILS for the same instance and from the same initial solution. Figure 10 shows the number of checkers not satisfied by the current solution at each iteration in a single execution of each search for an instance with 2000 nodes and a density of 0.5, while Fig. 11 considers an instance with 2500 nodes and the same density. The initial solutions were obtained with the construction heuristic. Figures 12 and 13 show, for the same executions of each search, the objective function value of the current solution at each iteration of the search.

As for the TSP, the number of iterations of DILS is larger than for the other methods. In the first iterations of the search, DILS moves to solutions with approximately three times the number of unsatisfied checkers compared to the initial solution. Then, DILS slowly converges and finds better local optima than both BILS and FILS.

Table 3 gives the average number of iterations for each method over 100 restarts. The number of iterations of DILS is again close to the double of the number of iterations of BILS in all instances considered, with an average ratio of 1.82.

8.2 Search performance

As for the TSP, we show results for max-cut and the 1-flip neighborhood using both a fixed time limit and a fixed target. Table 4 shows results of performing DILS, BILS and FILS for 60 s, executing each multi-start method ten times, starting from solutions generated by the construction heuristic described in Sect. 6.2.

The superior performance of DILS is clear. For 26 of the 30 instances, DILS obtained the best solution, whereas FILS obtained the best solution in three instances and BILS

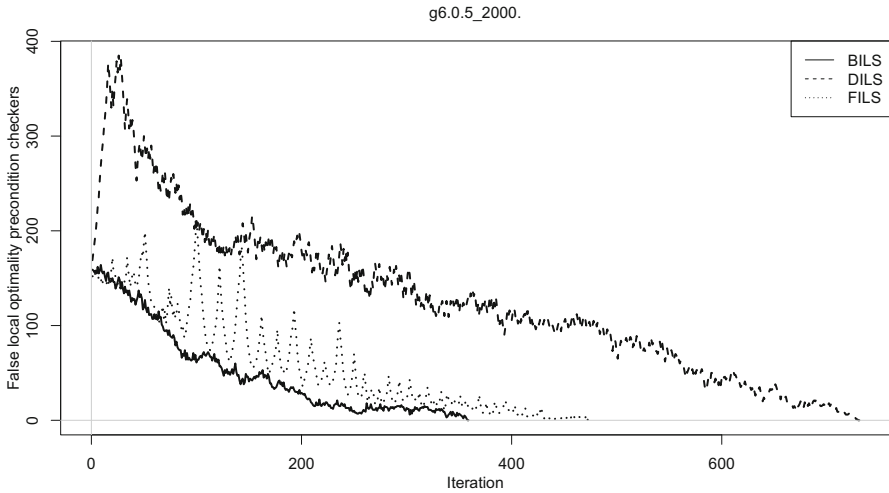


Fig. 10 Number of checkers not satisfied by the current solution for a instance with 2000 vertex and density 0.5, with the initial solution created by a constructive heuristic

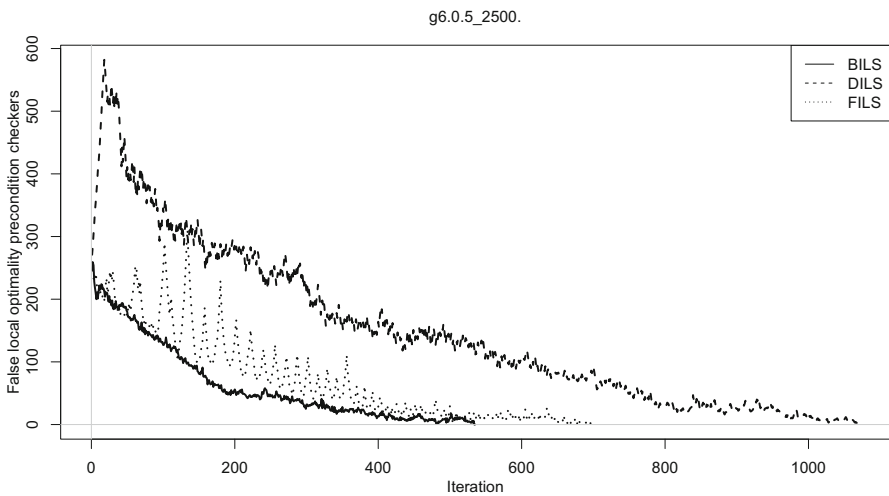


Fig. 11 Number of checkers not satisfied by the current solution for a instance with 2500 vertex and density 0.5, with the initial solution created by a constructive heuristic

obtained the best solution only for one instance. Here, the average behavior is even more in favor of DILS, as the method has the best average performance on all 30 instances, whereas the other two methods fail to obtain the best average performance on any instance. Considering the % gap to the best solution found for each instance within the experiment, the same dominance is revealed. Applying Welch’s t-test on the % gap and the chi-squared test on the number of best performances reveal that both of these differences are statistically significant at any reasonable significance level.

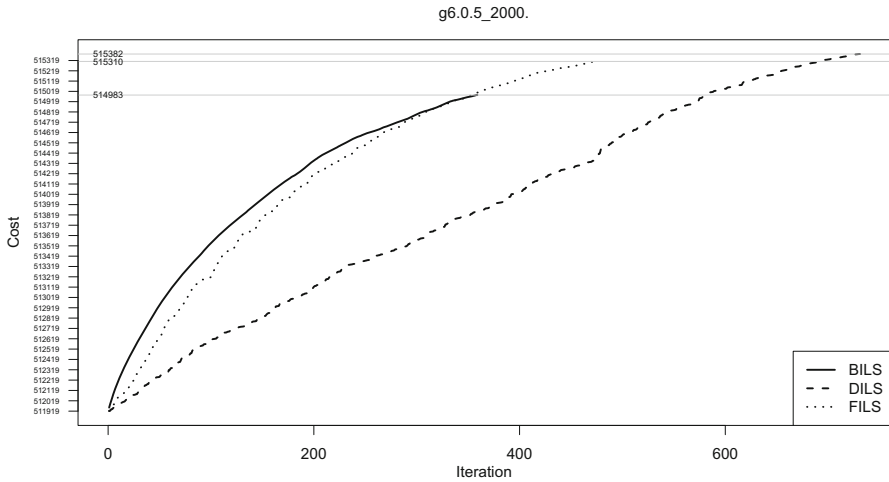


Fig. 12 Objective function value of the current solution for a instance with 2000 vertex and density 0.5, with the initial solution created by a constructive heuristic

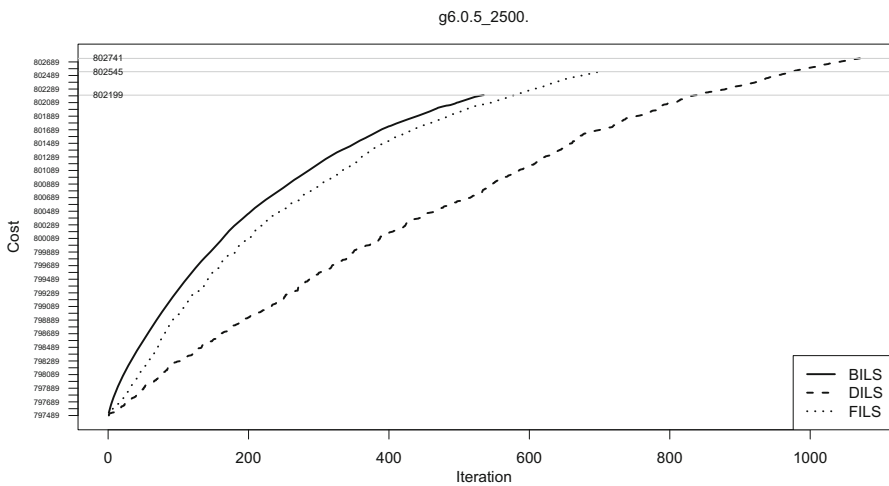


Fig. 13 Objective function value of the current solution for a instance with 2500 vertex and density 0.5, with the initial solution created by a constructive heuristic

In the second set of performance evaluation tests we produce time to target (TTT) plots. Figures 14, 15, and 16 show TTT plots for instances with 1500, 2000, and 2500 nodes. The density for each graph is 0.5. The overall time limit was set to 180 s. Three plots are shown for each instance, considering progressively harder targets. The hardest target is set to the best average objective function value, considering each method separately, in Table 4 minus 0.05% rounded to the nearest integer. The other two targets are 0.1% and 0.15% below the best average, rounded to the nearest integer.

The plots show that with easy targets and large graphs BILS and FILS outperform DILS. However, in these cases all the three algorithms reach the target within a few

Table 3 Number of iterations performed by each search.

	IVI	D	BILS	FILS	DILS	DILS/BILS
1	1500	0.45	364.10	402.00	628.40	1.73
2	1500	0.46	346.30	416.20	624.50	1.80
3	1500	0.47	349.40	400.50	620.80	1.78
4	1500	0.48	331.20	417.90	627.60	1.89
5	1500	0.49	320.40	379.70	596.30	1.86
6	1500	0.50	345.90	385.50	609.10	1.76
7	1500	0.51	344.90	410.90	601.30	1.74
8	1500	0.52	332.50	404.80	611.30	1.84
9	1500	0.53	356.40	413.20	609.70	1.71
10	1500	0.54	324.60	422.80	617.10	1.90
11	2000	0.45	517.00	530.30	887.40	1.72
12	2000	0.46	495.30	575.90	870.60	1.76
13	2000	0.47	491.20	585.00	894.10	1.82
14	2000	0.48	431.70	596.00	901.00	2.09
15	2000	0.49	430.80	554.90	835.90	1.94
16	2000	0.50	450.60	615.70	865.60	1.92
17	2000	0.51	505.50	578.30	876.60	1.73
18	2000	0.52	463.90	533.00	863.30	1.86
19	2000	0.53	469.70	574.40	876.40	1.87
20	2000	0.54	472.20	554.90	885.20	1.87
21	2500	0.45	658.80	747.10	1132.80	1.72
22	2500	0.46	654.00	757.00	1149.80	1.76
23	2500	0.47	644.20	751.20	1161.70	1.80
24	2500	0.48	572.90	696.70	1160.30	2.03
25	2500	0.49	672.90	798.10	1162.80	1.73
26	2500	0.50	611.80	757.20	1157.40	1.89
27	2500	0.51	633.90	698.30	1104.70	1.74
28	2500	0.52	626.10	751.10	1138.00	1.82
29	2500	0.53	634.30	725.70	1091.50	1.72
30	2500	0.54	569.90	727.80	1073.20	1.88

Average of 100 executions

seconds. On the other hand, when the target gets harder to achieve, the performance of DILS is much better than BILS and FILS.

9 Conclusions

In this work, we introduced delayed improvement local search (DILS), a local search technique based on the idea of selecting as the next solution the improving neighbor that has fewer characteristics of locally optimal solutions. To implement this idea we

Table 4 Results of performing DILS, BILS and FILS for 60s, 10 repetitions, starting from solutions generated by the constructive heuristic

	V	D	BILS		FILS		DILS	
			Avg. Z	Max. Z	Avg. Z	Max. Z	Avg. Z	Max. Z
1	1500	0.45	263644.1	263785	263717.2	263864	263875.1	263958
2	1500	0.46	268781.1	268869	268795.2	268982	268911.9	269020
3	1500	0.47	274362.1	274433	274381.1	274422	274547.7	274714
4	1500	0.48	280477.1	280542	280549.7	280631	280675.1	280770
5	1500	0.49	285535.6	285671	285574.3	285641	285690.0	285852
6	1500	0.50	290937.2	291142	291005.4	291163	291123.5	291256
7	1500	0.51	297168.7	297344	297202.7	297402	297381.4	297456
8	1500	0.52	302495.7	302613	302568.1	302792	302670.8	302711
9	1500	0.53	308096.4	308224	308148.7	308231	308292.8	308440
10	1500	0.54	314198.3	314277	314177.1	314263	314328.4	314463
11	2000	0.45	466453.5	466748	466489.7	466603	466667.2	466872
12	2000	0.46	475573.2	475762	475605.2	475724	475779.4	475882
13	2000	0.47	485711.8	485802	485858.8	486102	485986.6	486122
14	2000	0.48	495714.7	495772	495737.6	495827	495940.5	496075
15	2000	0.49	505281.9	505449	505363.5	505565	505466.2	505687
16	2000	0.50	515741.1	515807	515737.7	515919	515892.7	516056
17	2000	0.51	526140.2	526320	526128.6	526285	526280.7	526382
18	2000	0.52	535987.2	536151	536066.9	536221	536278.2	536502
19	2000	0.53	545706.9	545851	545746.8	545976	545853.0	545964
20	2000	0.54	556148.2	556475	556136.1	556238	556298.3	556474
21	2500	0.45	725663.7	725809	725692.0	725783	725898.6	726034
22	2500	0.46	740798.9	740860	740819.7	741013	740993.8	741058
23	2500	0.47	756634.5	756703	756705.9	756997	756765.7	756969
24	2500	0.48	771467.9	771665	771549.8	771732	771679.2	771845
25	2500	0.49	788344.5	788511	788364.1	788536	788547.9	788818
26	2500	0.50	803207.0	803526	803157.3	803322	803275.9	803572
27	2500	0.51	818493.9	818778	818460.7	818623	818571.5	818791
28	2500	0.52	834306.3	834597	834334.9	834421	834508.4	834836
29	2500	0.53	849186.7	849358	849241.3	849420	849404.1	849532
30	2500	0.54	865441.2	865644	865536.8	865694	865745.8	866126
No. of best			0	1	0	3	30	26
Avg. % gap			0.0	0.04	0.03	0.03	0.00	0.00
Std. % gap			0.02	0.02	0.01	0.02	0.00	0.00

The best obtained solution for each instance is highlighted in bold

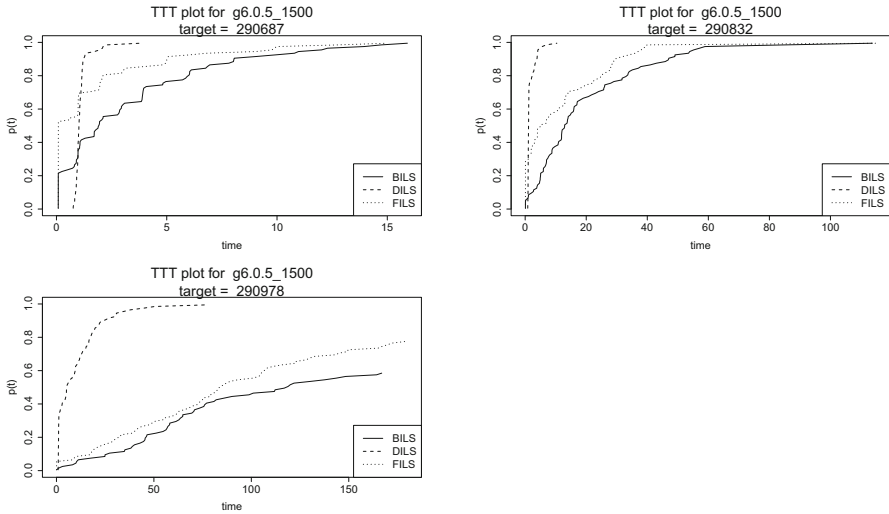


Fig. 14 TTT plots for a graph with 1500 vertex and 0.5 density, with easiest target on the top left and the hardest target on the bottom left

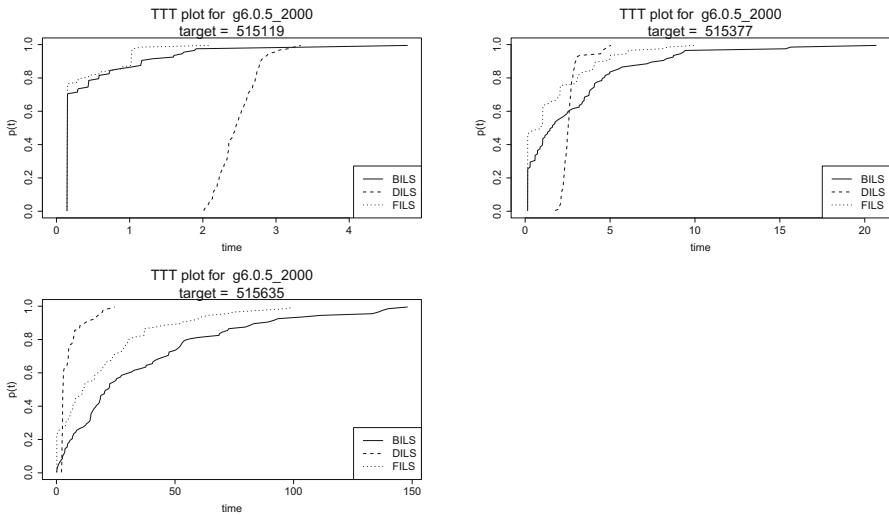


Fig. 15 TTT plots for a graph with 2000 vertex and 0.5 density, with easiest target on the top left and the hardest target on the bottom left

evaluated for each neighbor solution the compliance with local search inequalities that are always satisfied by locally optimal solutions.

Each inequality gives rise to a local optimality precondition checker. There is a one to one correspondence between a local optimality precondition checker evaluated as false and an unsatisfied local search inequality. In each iteration of DILS, the number of local optimality precondition checkers evaluated to false is computed for each

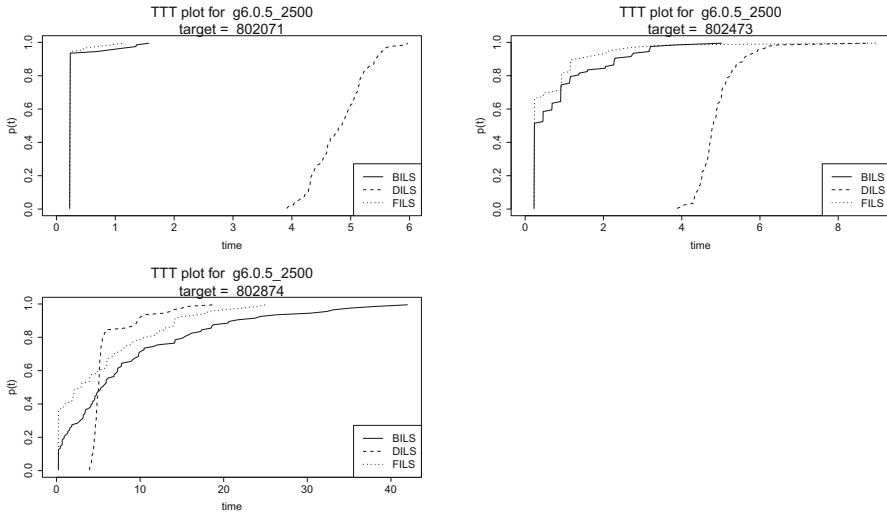


Fig. 16 TTT plots for a graph with 2500 vertex and 0.5 density, with easiest target on the top left and the hardest target on the bottom left

neighbor and this number along with the solution cost is used to determine the next current solution.

We implemented DILS for the 2-opt local search of the traveling salesman problem (TSP) and for the 1-flip local search of the max-cut problem using the local search inequalities from Lancia et al. (2015). Computational results show that, for the instances tested, DILS has a better performance than the classic local search implementations known as best improvement local search and first improvement local search when initial solution are constructed with greedy construction heuristics.

The efficacy of DILS depends on the availability of local optimality checkers for the neighborhood operators considered. If local search inequalities are not currently available for a neighborhood operator, local optimality checkers can be obtained by studying the properties of local optima for the neighborhood being used.

The performance of DILS also depends on the implementation of the test for the local optimality checkers. In this work we stored information about the neighboring solutions in previous iterations, which helped us to quickly recompute the number of checkers evaluated to false for each neighbor after each step of the local search. With better implementations, using more stored information and more complex data structures it might be possible to further improve the performance of DILS.

Future research includes applying DILS to other neighborhoods for the TSP such as node swap, for which Lancia et al. (2015) provided local search inequalities, and 3-opt. DILS can also be applied to other combinatorial optimization problems, in addition to the TSP and the max-cut. Finally, the idea of DILS to use local search inequalities may be extended to local search based metaheuristics such as tabu search, iterated local search, and variable neighborhood search. In each of these, being able to postpone the convergence to a local optimum, beyond what can be done using standard search components, may turn out to be very beneficial.

Acknowledgements The authors wish to thank the editor and the reviewers for their insightful comments and suggestions, which helped to improve this paper.

Funding Open access funding provided by Molde University College - Specialized University in Logistics.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aiex, R.M., Resende, M.G., Ribeiro, C.C.: TTT plots: a perl program to create time-to-target plots. *Optim. Lett* **1**(4), 355–366 (2007)
- Bertsimas, D.: Probabilistic combinatorial optimization problems. Ph.d thesis, Massachusetts Institute of Technology (1988)
- Černý, V.: Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *J. Optim. Theory Appl.* **45**(1), 41–51 (1985)
- Duarte, A., Sánchez-Oro, J., Mladenović, N., Todosijević, R.: Variable neighborhood descent. In: *Handbook of Heuristics*, pp. 341–367 (2018)
- Festa, P., Pardalos, P., Resende, M., Ribeiro, C.: Randomized heuristics for the max-cut problem. *Optim. Methods Softw.* **7**, 1033–1058 (2002)
- Garey, M., Johnson, D., Stockmeyer, L.: Some simplified NP-complete graph problems. *Theoret. Comput. Sci.* **1**(3), 237–267 (1976)
- Glover, F.: Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.* **13**(5), 533–549 (1986)
- Glover, F.: Tabu search-part i. *ORSA J. Comput.* **1**(3), 190–206 (1989)
- Glover, F.: Tabu search-part ii. *ORSA J. Comput.* **2**(1), 4–32 (1990)
- Hansen, P., Mladenović, N.: First vs. best improvement: an empirical study. *Discrete Appl. Math.* **154**(5), 802–817 (2006)
- Johnson, D.S., McGeoch, L.A.: The traveling salesman problem: a case study in local optimization. In: Aarts, E.H.L., Lenstra, J.K. (eds.) *Local Search in Combinatorial Optimization*, pp. 215–310. Wiley, Chichester (1997)
- Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* **220**(4598), 671–680 (1983)
- Kumar, S.N., Panneerselvam, R.: A survey on the vehicle routing problem and its variants. *Intell. Inf. Manag.* **4**, 66–74 (2012)
- Lancia, G., Rinaldi, F., Serafini, P.: Local search inequalities. *Discret. Optim.* **16**, 76–89 (2015)
- Laporte, G.: The traveling salesman problem: an overview of exact and approximate algorithms. *Eur. J. Oper. Res.* **59**(2), 231–247 (1992)
- Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search: framework and applications. In: Gendreau, M., Potvin, J.Y. (eds.) *Handbook of Metaheuristics*, pp. 363–397. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-1665-5_12
- Mladenović, N., Hansen, P.: Variable neighborhood search. *Comput. Oper. Res.* **24**(11), 1097–1100 (1997)
- Nowicki, E., Smutnicki, C.: A fast taboo search algorithm for the job shop problem. *Manag. Sci.* **42**(6), 797–813 (1996)
- Pekny, J.F., Miller, D.L.: A staged primal-dual algorithm for finding a minimum cost perfect two-matching in an undirected graph. *ORSA J. Comput.* **6**(1), 68–81 (1994)
- Prandtstetter, M., Raidl, G.R.: An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *Eur. J. Oper. Res.* **191**(3), 1004–1022 (2008)

- Rego, C., Gamboa, D., Glover, F., Osterman, C.: Traveling salesman problem heuristics: leading methods, implementations and latest advances. *Eur. J. Oper. Res.* **211**(3), 427–441 (2011)
- Reinelt, G.: TSPLIB—a traveling salesman problem library. *ORSA J. Comput.* **3**(4), 376–384 (1991)
- Solnon, C., Cung, V.D., Nguyen, A., Artigues, C.: The car sequencing problem: overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *Eur. J. Oper. Res.* **191**(3), 912–927 (2008)
- Voudouris, C.: Guided local search for combinatorial optimisation problems. Ph.d thesis, University of Essex (1997)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.