Iterated Greedy

18

Thomas Stützle and Rubén Ruiz

Contents

T. Stützle (✉)
IRIDIA, Université Libre de Bruxelles (ULB), Brussels, Belgium
e-mail: stuetzle@ulb.ac.be

R. Ruiz
Grupo de Sistemas de Optimización Aplicada, Instituto Tecnológico de Informática, Ciudad Politécnica de la Innovación, Edificio 8G, Acc. B. Universitat Politécnica de València, València, Spain
e-mail: rruiz@eio.upv.es

R. Martí et al. (eds.), *Handbook of Heuristics*,
https://doi.org/10.1007/978-3-319-07124-4_10

Abstract

Iterated greedy is a search method that iterates through applications of construction heuristics using the repeated execution of two main phases, the partial destruction of a complete candidate solution and a subsequent reconstruction of a complete candidate solution. Iterated greedy is based on a simple principle, and methods based on this principle have been proposed and published several times in the literature under different names such as simulated annealing, iterative flattening, ruin-and-recreate, large neighborhood search, and others. Despite its simplicity, iterated greedy has led to rather high-performing algorithms. In combination with other heuristic optimization techniques such as a local search, it has given place to state-of-the-art algorithms for various problems. This paper reviews the main principles of iterated greedy algorithms, relates the basic technique to the various proposals based on this principle, discusses its relationship with other optimization techniques, and gives an overview of problems to which iterated greedy has been successfully applied.

Keywords

Stochastic local search · Metaheuristics · Iterated greedy · Greedy methods · Local search · Constructive search

Introduction

Many effective algorithms for $\mathcal{NP}$-hard combinatorial optimization problems rely on the efficient, repeated execution of some simple, underlying mechanisms. A common example is perturbative search methods that iterate over neighborhood searches or iteratively apply underlying iterative improvement procedures. Examples of such methods include simulated annealing, tabu search, iterated local search, memetic algorithms, or dynamic local search [41,48]. In fact, simulated annealing, tabu search, or dynamic local search at each step explore the neighborhood of a current solution, while iterated local search and memetic algorithms can be seen as iterating across repeated applications of improvement algorithms. Less frequently, stochastic local search (SLS) methods make use of the iterative application of solution construction algorithms. Examples for these latter methods are ant colony optimization (ACO) [23, 24], greedy randomized adaptive search procedures (GRASP) [30, 31], or the pilot and rollout method [11, 27].

In this chapter, we review another SLS method that relies on the iterative application of solution construction procedures: iterated greedy. The method builds a sequence of solutions by iterating through phases of (partial) solution destruction and subsequent reconstruction of a complete candidate solution. A first complete solution in this sequence is generated by some constructive method. Then the following three steps are iteratively executed. First, components of a complete candidate solution s are removed, resulting in a partial solution s_p. Second, starting

from s_p, a complete candidate solution s' is rebuilt. Third, an acceptance criterion decides whether to continue this process from s or s'.

Iterated greedy has a clear underlying principle, and it is generally applicable to any problem for which constructive methods can be conceived. As such, iterated greedy is clearly a general-purpose method. Iterated greedy is a rather simple method that needs typically only short development times, especially if already a constructive heuristic is available. Iterated greedy provides also a rather simple way of improving over the single application of a constructive method, and for various problems very high-quality solutions are generated. Additionally, basic versions of iterated greedy do only incur few main parameters, and their impact on the search process is rather intuitive to understand. All these reasons make iterated greedy a desirable technique for developers of heuristic algorithms.

Given the simple underlying principle, it is maybe also not surprising that the method that we here call iterated greedy has been (re-)discovered and applied a large number of times under different names by different authors (including ourselves). Algorithms that rely to a significant extent on the same underlying principle have been given names such as simulated annealing [50], evolutionary heuristic [62], iterative flattening [17], ruin-and-recreate [95], iterative construction heuristic [86], large neighborhood search [96], or, as here, iterated greedy [48, 92]. We will review these different developments and other related procedures in section "IG Applications: Historical Development."

Despite the possible confusion that may arise for the reader due to the different names and the in part different views on the method, we want to stress that the really important aspect is the principle that underlies all these algorithms. In fact, in all these proposals a repeated usage of constructive methods is made that start from some intermediate, partial candidate solutions. This principle is a generic one of a potential large utility, and it should be understood as one of the basic principles that can be used to develop optimization algorithms. This basic principle may also be only one of the principles that is used in the development of a hybrid optimization algorithm that combines elements from different techniques. For example, several algorithms that make use of the iterated greedy principle include also a local search phase that may improve the solutions generated by the constructive mechanisms [92].

The structure of the chapter is as follows. In section "Iterated Greedy" we review the basic principles of iterated greedy algorithms. Next, in section "Some Simple Examples of Iterated Greedy Algorithms," we give some concrete examples of iterated greedy algorithms. In section "Case Study: Iterated Greedy for Flow Shop Scheduling" we give some results of an experimental study that discusses the main trade-offs in the design of an iterated greedy algorithm for the permutation flow shop problem. We then present other algorithms that make use of the same principle as iterated greedy in section "IG Applications: Historical Development." The relationship of iterated greedy to some other methods is discussed in section "Relationship to Other Approaches." References to some noteworthy applications of iterated greedy are given in section "Applications," and we conclude in section "Conclusions."

Iterated Greedy

Greedy Construction Heuristics

Constructive algorithms build candidate solutions to optimization or decision problems step by step, starting from an empty solution. At each step, they add a solution component to the current partial solution and repeat these steps until a complete candidate solution is obtained. Commonly, constructive algorithms use a heuristic function that estimates for each solution component the benefit of including it into a partial candidate solution. A baseline construction algorithm is formed by so-called greedy (constructive) algorithms that at each step add a solution component for which the value of the heuristic function is the best (see also Fig. 1). If more than one solution component has the same best heuristic value, a tiebreaking criterion is used to decide which solution component is actually added; in the simplest case, this tiebreaking is done uniformly at random, but it also may be done by a secondary heuristic function.

Greedy construction heuristics are frequently used when tackling combinatorial optimization problems due to a number of reasons. First, greedy construction heuristics are rather fast, and at the same time they generate solutions that are usually much better than those generated uniformly at random or by a randomized but heuristically biased construction. Second, these algorithms are often used to seed (perturbative) local search methods such as iterative improvement algorithms; more sophisticated SLS methods such as tabu search and simulated annealing; or population-based methods such as memetic algorithms. In the latter case, typically some members of the population are generated by greedy constructive methods, while others may be randomly generated. Seeding perturbative local search methods with solutions from greedy construction algorithms can incur advantages such as improved quality of local optima, faster identification of local optima, and a better trade-off between computation times and solution quality, that is, better anytime behavior [114]. Third, sometimes one can prove guarantees on the quality of the solutions that are generated in the worst case, leading to so-called approximation algorithms. Often, the best provable guarantees that can be obtained even for more complex SLS algorithms are the guarantees that directly stem from those of the initial greedy construction. Fourth, for various polynomially solvable problems, greedy algorithms are also guaranteed to generate optimal solutions, the Kruskal algorithm for minimum spanning trees being a well-known example. However, for $\mathcal{NP}$-hard problems, this is not the case. Finally, they build the basis for a

Fig. 1 Algorithmic outline of a greedy heuristic

```
procedure Greedy Constructive Heuristic:
    s = ∅
    while s is not a complete solution do
        choose a best rated solution component c
        s = s + c
    end while
end
```

number of other methods such as GRASP [30, 31], ACO [23], or squeaky wheel optimization [52].

One straightforward way to improve over the generation of a single greedy solution is in some cases the repeated application of a greedy heuristic to generate a variety of different candidate solutions and then to choose the best one. Obviously, repetition in this sense is only reasonable if in the construction process different solutions can be generated. For example, for the well-known nearest neighbor heuristic for the traveling salesperson problem (TSP), n distinct nearest neighbor tours may result (assuming no random tiebreaking is done); for each of the n possible cities that may be chosen (randomly) as the initial city for the solution construction, a different tour may result. However, in other cases where the greedy construction is fully deterministic, additional randomization of the construction process, as proposed in the semi-greedy heuristics [45] and in GRASP [30, 84], may be required to generate different solutions.

Repeated construction of solutions also has inherent disadvantages. Constructing a full solution is relatively time-consuming as especially the initial construction steps require a large amount of computation when compared to later construction steps. Furthermore, no information is taken from one solution construction to another one, and thus such a repeated construction does not exploit knowledge gained from previous solutions. A method that alleviates these problems and that allows to invest, in principle, arbitrary computing times to generate different solutions by constructive heuristics is iterated greedy.

Iterated Greedy Framework

The main principle of iterated greedy is to iterate over (greedy) construction methods by first generating a complete candidate solution and then cycling through a main loop that consists of two main steps. In the first step, some solution components are removed from the current complete candidate solution s to result in some intermediate partial candidate solution s_p. We call this the *destruction step*. In the next step, starting from s_p, a construction heuristic is used to generate a new complete solution s'. We call this step the *construction step*. An acceptance test then decides from which of the two solutions, s or s', the next destruction step applies. While in the simplest case, the acceptance test may accept only improved solutions, other choices may lead to more search diversification and, thus, to possibly better results when many iterations of the iterated greedy algorithm are done.

An algorithmic outline of an IG algorithm is given in Fig. 2. It starts by first generating an initial candidate solution using a procedure GenerateInitialSolution and then iterates through a main loop that consists of the application of the three procedures Destruction, Construction, and AcceptanceCriterion. Note that the construction procedures used in GenerateInitialSolution and Construction may be different and, hence, may also use different greedy heuristics. In the simplest case, however, they may be the same procedures.

Fig. 2 Algorithmic outline of Iterated Greedy (IG)

```
procedure Iterated Greedy
    s_0 = GenerateInitialSolution
    repeat
        s_p = Destruction(s*)
        s' = Construction(s_p)
        s* = AcceptanceCriterion(s*, s')
    until termination condition met
end
```

A simple default version of an iterated greedy algorithm could use the following choices. As constructive heuristic one may take one that is either already available or implement a known state-of-the-art constructive heuristic. The solution destruction may delete some randomly chosen solution components, where the number d of solution components to be removed is a parameter of the algorithm. As acceptance criterion, one may force the cost to decrease by only accepting improved or equal quality candidate solutions.

The iterative process around construction heuristics gives IG some specific advantages when compared to the repeated construction of complete candidate solution from scratch. In fact, by starting from a partial solution, one may reduce significantly the time necessary to generate a new candidate solution as less constructive steps need to be done, and the time per construction decision is also reduced as the number of available solution components to choose from is smaller the larger the partial solutions. In addition, through the potential bias exerted by the acceptance criterion, the search process can more easily intensify around the best solutions found in the search process.

When trying to develop a more performing version of an iterated greedy algorithm, many different options for each of the specific operators may be taken into account. Some of the main relevant issues to be considered will be discussed in the following. Other ideas will be discussed also later when considering the relationship of iterated greedy to other methods or when discussing applications of iterated greedy algorithms.

Destruction There are a number of different possible choices for the solution destruction. A first consideration concerns the number of solution components that should be removed, as defined by a parameter d. The extreme settings would correspond to removing only a single component, that is, $d = 1$ or all of them. Even if one may argue that the resulting algorithms should be considered as iterated greedy algorithms, these parameter settings would not correspond to the main ideas underlying iterated greedy. The first case would be akin to a randomized local search, while the latter be akin to a repeated application of a construction heuristic. (We discuss these relationships in more depth in section "Relationship to Other Approaches.") Intermediate values of d result in a trade-off between search intensification and diversification: removing a large number of solution components allows to jump to rather distant solutions in the construction phase, while removing few components leads to a more localized search.

Another choice is whether to leave the number of solution components to be removed fixed or variable during the algorithm run. In the case d is left variable, a scheme of how to adapt its value is required. If the value of d is left fixed, it requires proper tuning. Possibilities for varying the value of d could be to modify this value randomly within some interval every few iterations, to choose the value according to a scheme similar as those introduced for variable neighborhood search [44] or to adapt the value of d at computation time, exploiting ideas of reactive search [9].

Once it is known how many components are to be removed, which ones should be chosen? There are a number of possible answers to this question. Intuitively, a randomized destruction is preferable over deterministic choices to reduce the danger of cycling. If a stochastic destruction is used, the simplest case is to choose components uniformly at random. More involved could be choices that take into account cost measures on the components or the partial solutions that remain, thus introducing a bias in the choice. In that case, solution components that have a strong contribution to the cost of a solution would then have a larger probability of being removed than low cost components. Alternatively, one may use lower bounds on the partial solutions resulting after having removed a component: the smaller the lower bound, the higher the probability.

Construction One of the crucial ideas of IG is that restoring a fully specified solution is done by some form of (greedily biased) constructive mechanism. Hence, the usage of a constructive mechanism is essential. Using a random perturbation (corresponding to a move to a random solution in a large neighborhood) is more in the spirit of reduced variable neighborhood search than in the spirit of IG.

The naming iterated greedy suggests that the construction heuristics used in the algorithm are greedy construction heuristics and typically deterministic (modulo random tiebreaking). In fact, in many implementations of iterated greedy algorithms, this is also the case. However, we want to emphasize here that this need not necessarily be the case as, in principle, any suitable constructive mechanism that starting from some partial candidate solution s_p can generate a complete candidate solution – be it deterministically greedy, probabilistically greedy, or else – may be used as the underlying construction mechanism in an iterated greedy algorithm.

Among the construction rules, one may distinguish between *adaptive* heuristics and *static* ones. In the first case, the heuristic value assigned to a particular choice in the solution construction depends on the partial solution. Typically, adaptive construction heuristics will result in better quality solutions than static ones; however, this improved solution quality is often reached at the cost of higher computation times.

In the simplest case, the solution construction is done following a deterministic construction heuristic – deterministic except of maybe random tiebreaking. Depending on the construction mechanism, it is possible to apply a deterministic rule: by applying stochastic destruction, either the order in which solution components are added is modified or, if adaptive construction heuristics are used, their heuristic evaluation.

Such basic considerations about solution construction in iterated greedy algorithms can be extended in many natural ways by adopting techniques that have been proposed in other methods. One may take inspiration from other constructive methods and use randomized selection in candidate lists built at each construction step as in GRASP algorithms, use biased constructive decisions exploiting past experience such as the pheromone trails in ACO, or use prohibitions by avoiding adding again solution components that have been removed in the solution destruction, taking ideas from tabu search. Another option may be to choose among different possible construction rules and use ideas from the hyperheuristics community for this task [15]. In fact, all kind of methods and techniques that are compatible with the idea of constructing solutions may be adopted within iterated greedy algorithms if this seems promising, making iterated greedy in this sense also a very flexible technique.

Acceptance criterion. The acceptance criterion has a strong influence on the diversification/intensification behavior of an IG algorithm. On the extreme cases are the possibilities of accepting any new solution independent of its solution quality or to only accept solutions that improve over the previous one. There are many intermediate choices such as occasionally accepting worse solutions or allowing backtracks to previously seen solutions. A popular choice when accepting worse solutions is to use acceptance criteria from simulated annealing such as the Metropolis criterion: if a new solution is better or equal, it is accepted; otherwise it is accepted with a probability $\exp\{(f(s) - f(s'))/T\}$, where T is a parameter called temperature. More elaborated approaches might probably make use of short-term memory as in tabu search. The acceptance criterion, whatever it might be, does not need to be applied at every iteration, i.e., a given incumbent solution can be destructed and reconstructed a number of times before deciding on its value. For the choice of the acceptance criterion, very much the same issues arise as in iterated local search [82], and an appropriate choice may be crucial to an IG algorithm's performance.

Hybridization with local search. Iterated greedy algorithms can form directly the basis of hybrid algorithms that combine various search mechanisms. In fact, for constructive heuristics a natural extension is to improve the generated solutions by the application of a (perturbative) local search method, in the simplest case this being an iterative improvement algorithm. Such an extension is also straightforward to be adopted in an iterated greedy algorithm and actually has been done in a number of such approaches. This extension results in an outline of iterated greedy as given in Fig. 3. With this additional local search phase, the IG algorithm also strongly resembles iterated local search (ILS) algorithms [82]. In fact, the destruction and construction phases implement a solution perturbation in the ILS sense. However, a minor difference is that ILS often makes use of perturbations that are randomly chosen from some large neighborhood, while in IG algorithms an underlying constructive method is exploited. More importantly, IG can also reach very high performance when used without the local search phase, which is not necessarily

Fig. 3 Algorithmic outline of an IG with an additional local search step

```
procedure Iterated Greedy with local search
    s_0 = GenerateInitialSolution
    s* = LocalSearch(s_0)
    repeat
        s'_p = Destruction(s*)
        s' = Construction(s'_p)
        s' = LocalSearch(s')
        s* = AcceptanceCriterion(s*, s')
    until termination condition met
end
```

true for ILS algorithms. More on relations of IG to other methods is given in section "Relationship to Other Approaches."

In any case, for an algorithm to qualify as an iterated greedy algorithm, it is necessary that one can distinguish between a clearly available construction mechanism and a clear destruction mechanism that are repeatedly applied in an alternating order. An acceptance criterion may be used in an explicit way or in an implicit way; the latter is the case, for example, if every new solution is accepted and no mentioning of an acceptance criterion is done. In that case, the acceptance criterion would correspond to a "random walk-type" acceptance criterion used sometimes in ILS algorithms [82].

Some Simple Examples of Iterated Greedy Algorithms

Let us consider a few examples of constructive heuristics for basic combinatorial problems that will serve throughout the chapter for illustrating various details of IG algorithms. We consider here three examples for the traveling salesman problem (TSP), the set covering problem (SCP), and the permutation flow shop problem (PFSP). We first shortly introduce the problem and then describe a basic constructive heuristic and basic IG algorithms.

TSP Example

Traveling salesman problem (TSP). The TSP is given a graph $G = (N, E)$ where N is the set of $n = |N|$ nodes and E is the set of edges that fully connects the nodes. To each edge (i, j) is associated a distance d_{ij}. Here we assume that the distance matrix is symmetric, that is, we have $d_{ij} = d_{ji}$ for all $(i, j) \in E$; this type of TSP instances are called symmetric and are among the most widely studied types of TSP instances. The objective in the TSP is to determine a Hamiltonian cycle of minimal length. Such a cycle can be represented by a permutation $\pi = \langle \pi(1), \ldots, \pi(n) \rangle$, where $\pi(i)$ is the node index at position i. The objective function to be minimized is

$$\min_{\pi \in \mathcal{S}} \; d_{\pi(n)\pi(1)} + \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} \tag{1}$$

where $\mathcal{S}$ is the search space consisting of the set of all permutations.

Probably the best known constructive heuristic for the TSP is the nearest neighbor heuristic. It chooses randomly a first node to start from to obtain a partial tour $\langle \pi(1) \rangle$. Then at each step it appends to the partial tour $\langle \pi(1) \dots \pi(l) \rangle$, a still unvisited node that has minimum distance to $\pi(l)$. Once all nodes are visited, the tour is completed by closing the tour and going back to $\pi(1)$. The nearest neighbor heuristic usually contains subtours that may be close to optimal ones but also contains long edges that are added often toward the end of the construction. Empirically, for Euclidean two-dimensional TSP instances, the nearest neighbor heuristic generates tours that are about 20–40% above optimal (see [51]).

One possibility to build an iterated greedy algorithm on top of the nearest neighbor heuristic would be to remove a subtour of d consecutive cities of a tour, resulting in one subtour of $n - d$ cities and to restart the nearest neighbor heuristic from it. Another possibility would be to remove randomly cities, reconnect the remaining subtours, and then restart the nearest neighbor heuristic. However, it is unclear what the performance of such an algorithm would be.

Another simple constructive heuristic for the TSP is the random insertion heuristic (RIH). It starts by choosing randomly two nodes that are ordered arbitrarily. At each construction step, a next node is chosen randomly and inserted into a position in the current path such that the cost increase is minimum. The RIH is one of simplest heuristics for the TSP; however, it is among the best performing constructive algorithms for the TSP [51].

IG can be used on top of the RIH as follows. The first solution is constructed by RIH. The procedure Destruct would remove l nodes that are chosen uniformly at random. Next, Construct adds the removed nodes using again the same rules as in the RIH. Finally, AcceptanceCriterion could be chosen to only accept improved solutions. This means that the solution destruction would be applied to the best tour found so far. Different choices for iterated greedy algorithms based on such steps have also been experimentally examined by [95].

SCP Example

Set covering problem (SCP). The set covering problem (SCP) is given a set $A = \{a_1, \dots, a_n\}$ of items and a set $B = \{B_1, \dots, B_m\}$ of subsets of elements of A that covers A; in other words, we have that for each $B_i \subseteq A$ and $\bigcup_{i=1}^{m} = A$. A set B_i covers an item a_j if $a_j \in B_i$. Each set B_i has a cost of c_i. The objective in the SCP is to find a subset C of the sets in B that covers each element in A and that is of minimal total cost.

For the SCP, constructive heuristics differ in the choice of the heuristic function that is used. A standard heuristic function is to compute the cover value of the sets in B, which is defined as $\gamma_i = c_i / b_i$, where b_i is the number of items that would be covered when adding subset B_i to a current partial cover. Note that when starting with an empty solution, that is, none of the subsets B_i is chosen to be in the cover C, we have that $b_i = |B_i|$. However, each time a subset is chosen, b_i needs to be updated taking into account the items already covered in a partial cover C_p. The cover value is an example of an adaptive heuristic, where the heuristic values depend on the partial solution already generated. Adaptive heuristics require higher

computation times than their corresponding static heuristics, which do not update heuristic values in dependence of the partial solution, but typically lead also to better quality solutions. Once a complete cover is obtained, some of the subsets may have become redundant if all the items they cover are also covered by other subsets; removing such redundant subsets then improves the solution cost of the generated cover. Note that the removal of redundant subsets is not yet a destruction step as through the removal of redundant subsets the candidate solution remains a complete cover. The destruction step then may remove a number of subsets from a complete solution, resulting in a partial cover C_p, from which again the construction heuristic starts.

A first IG algorithm for the SCP has been proposed by [50]. They construct the first solution using a simple greedy construction heuristic [7] that at each step first selects a random, not covered item and adds the least cost subset that covers that item. Once a complete cover is found, redundant columns are removed. The solution destruction removes $k_1 \cdot |C|$ subsets that are chosen uniformly at random; here $0 < k_1 < 1$ is a parameter, and $|C|$ is the number of subsets on the cover and the cover size. The solution construction uses the heuristic based on cover values explained above. (The iterated greedy algorithm by [50] is, hence, a first example where the constructive heuristics for generating the initial solution and for extending partial solutions in the main loop differ.) However, the subsets to be considered are limited to a candidate set that comprises all those subsets that have a cost less than $k_2 \cdot \max\{c_i | i \in C\}$, where $k_2 > 0$ is a parameter that influences the size of the candidate set and C is the current cover. At each step of the construction, a subset with a minimum cover value is added to the current partial solution, breaking ties uniformly at random. Once a complete cover is obtained, redundant columns are removed as explained above. The acceptance criterion of a new cover C' is based on the Metropolis condition that is frequently used in simulated annealing algorithms. This simple IG algorithm for SCP obtained very high performance improving over several earlier proposed SCP heuristics.

PFSP Example

Permutation flow shop scheduling problem (PFSP). In the PFSP, n jobs have to be scheduled on m machines. All jobs visit the machines in the same order, each job having an operation at each machine. Time p_{ij} denotes the nonnegative, known, and deterministic processing time that job j needs on machine i. In the PFSP the same processing sequence of the jobs is maintained throughout all machines, and hence the processing sequence is obtained as a permutation of the jobs. The standard objective is to minimize the completion time of the last job in this order, which is also known as makespan ($C_{\max}$). The PFSP arises in many practical situations as it is common to have production lines where machines are disposed in series. The PFSP is a thoroughly studied problem in the scheduling literature with literally hundreds of papers published each year. The minimization of the makespan is the most used criterion in the literature, but not so in practice [36]. Some reviews on the PFSP are [35, 47, 91].

A popular and high performing constructive heuristic for the PFSP is the NEH heuristic [68], named after the initials of the last names of the paper's authors. NEH

is an insertion heuristic, a kind of heuristics that iteratively extends a permutation by inserting at each step one new element into the current partial solution. The NEH heuristic first computes for each job j its sum of the processing time $P_j = \sum_{i=1}^{n} p_{ij}$ and then orders them according to nonincreasing P_j values, resulting in a sequence $\phi(1) \ldots \phi(n)$. The first two jobs according to that order are taken, their two possible sequences $\langle \phi(1)\phi(2) \rangle$ and $\langle \phi(2)\phi(1) \rangle$ are evaluated, and the better of the two is adopted. Then at each step l, the job $\phi(l)$ is considered and tentatively inserted in all possible l positions in the current partial solution $\pi(1) \ldots \pi(l-1)$. Among these tentative insertions, the one resulting in the least increase of the makespan is taken. These steps are repeated until all jobs are inserted. Note that there might be two levels of ties, both at the ordering of the P_j values and at the insertion phase. The NEH heuristic has a computational complexity of $O(n^3m)$ which is lowered to $O(n^2m)$ using the efficient implementation of [97]. There is a rich literature of methods that propose variants of the NEH heuristic, mainly proposing mechanisms to break ties or reinsertions. As shown by [91] and more recently by [32], the NEH is a state-of-the-art constructive heuristic for the PFSP.

A simple IG algorithm for the PFSP has been proposed by [92]. It is based on insertion heuristics such as NEH. In fact, this iterated greedy algorithm uses NEH to construct an initial solution. At each destruction step, it removes a number of jobs that are chosen uniformly at random from the current permutation. In the construction step, these jobs are then reinserted in the same order in which they have been removed. For the insertion of the jobs, the same procedure as described for the NEH heuristic is followed. The acceptance criterion accepts a new candidate solution using the Metropolis condition known from simulated annealing algorithms but with the temperature T set to a constant value. The performance of this simple IG algorithm was very good, outperforming many metaheuristic algorithms for the PFSP. When combined with a local search phase, the proposed IG algorithm was shown to be a new state-of-the-art algorithm for the PFSP [92]. The IG implementation of [32] has given some further improvements of the above presented iterated greedy algorithm. Currently, the top-performing iterated greedy algorithm for the PFSP is the variant of [26], which additionally adds a local search on the partial solution that is obtained after the destruction step. As a matter of fact, this IG method has shown to outperform other more recent, and arguably much more complex, approaches.

Case Study: Iterated Greedy for Flow Shop Scheduling

The development of an effective iterated greedy algorithm requires the algorithm designer to choose which of the various possible implementation choices to take. In this section, we exemplify the development of an iterated greedy algorithm for the PFSP and study the impact specific alternative choices have on iterated greedy performance. Here we examine the impact of various alternative choices for the iterated greedy algorithm by [92] that we presented in the previous section.

In particular, we consider the following design choices in our experimental analysis.

Initial solution. It may seem advisable to start from an as good initial solution as possible. For the PFSP, this would be the NEH heuristic [32, 91] using the accelerations of [97]. However, at least for some instances, it is not clear whether a random or heuristic initialization of iterated greedy provides the best results. In fact, [81] study cases in which only for hard instances and in short CPU times it is advisable to use NEH (or extensions of the NEH) to obtain competitive results.

Destruction strength. A first and foremost decision to be taken is how many elements of a complete candidate solution should be removed. Here, we consider different fixed values for this parameter d, although it may also be interesting to consider schemes of how to vary the value of d at run-time.

Destruction type. The type of destruction determines which components of the incumbent solution are removed. In the simplest case, one may remove components of a solution uniformly at random. Alternatively, one may remove blocks of consecutive elements. In this case, a random block of d jobs could be removed from the sequence. Note that for a same solution π and a block-based destruction, there are only $n - d + 1$ possible choices for the destruction move. These two latter possibilities will be examined here.

However, one may consider a biased destruction, where components of the incumbent solution may be chosen randomly but in a biased way. For the PFSP, for example, jobs generating a large idle time at machines before or after their processing might be disrupting the sequence and are therefore likely to have been misplaced. We leave the study of such a destruction operator for future work.

Construction type. Similarly to the type of destruction move, different ways of how to construct a solution could be examined. The default choice in the iterated greedy algorithm is to insert jobs using the NEH heuristic. As an alternative, we consider a random insertion of the removed jobs, which could be used if no efficient greedy heuristic is available. However, in this case, the difference between iterated greedy and ILS becomes somewhat blurry, especially when additional local search is used.

Acceptance criterion. The acceptance criterion has a direct impact on the balance between intensification and diversification of the search. A simple idea is to accept only better quality solutions, while alternatively one may use the Metropolis condition that occasionally also accepts worse candidate solutions.

Acceptance iterations. Instead of applying the acceptance criterion at each iteration, one may apply an acceptance criterion only each l iterations. This would correspond to accepting for a few iterations every new candidate solution that is generated and only applying the acceptance test after each sequence of l steps.

Local search. Finally, it is well known that local search can have a tremendous impact on the quality of the results achieved, even though iterated greedy algorithms may reach high-quality solutions even without local search. Hence, it may be worthwhile to test the impact of an additional local search phase.

Table 1 Summary of the factors and the levels studied in the experimental analysis

Factor	Abbreviation	Level one	Level two
Initial solution	*Initialization*	NEH	Random
Destruction strength	*Destruct*	4	6
Destruction type	*Destruction_T*	Random	Block
Construction	*Reconstruction_T*	NEH	Random
Acceptance criterion	*Acceptance_C*	SA	Descent
Acceptance iterations	*Iterations_Acc*	1	5
Local search	*LS*	No	Yes

In order to study the different design and implementation alternatives, we have carried out a design of experiments (DOE) approach [67] where the previous factors are analyzed. Seven factors are coded and tested at two levels each, as summarized in Table 1. As a result, a total of $2^7 = 128$ combinations are to be tested in a full factorial experimental design. However, we use a half fractional design, which has most of the power of a full factorial one but requires only half of the runs. We use a 2^{7-1}_{VII} design, which has a high resolution VII: interactions between four factors are aliased (confounded) with interactions between three factors, interactions between five factors aliased with interactions between two factors, and so on. As a result, we can safely study the experimental data since it is highly unlikely that two high-level interactions might be significant. More elaborated techniques for the analysis and calibration of algorithms than the simple exploratory analysis we perform here have been published elsewhere [8]. Each studied combination has been tested on 30 of the hardest instances from Taillard's benchmark (99). This benchmark is composed of 12 groups of 10 instances, each ranging from 20 jobs and 5 machines to 500 jobs and 20 machines. The instances of one particular group are denoted as $n \times m$. For the tests we pick three of the hardest groups, namely, 50×20, 100×20, and 200×20. We are interested in these instances since for most of the remaining problems, the optimum solution is already known, and today's state-of-the-art methods are capable of obtaining near optimum solutions.

Each combination is run five independent times (replicates) with each instance. We also control the number of jobs (n) as a blocking factor with three levels in the experiment. Therefore, the experimental design contains $2^{7-1} \cdot 3 = 192$ treatments, $192 \cdot 10 = 1\,920$ experimental units (for ten instances), and $1\,920 \cdot 5 = 9\,600$ experiments (for five replicates).

The basic iterated greedy algorithm along with all studied alternatives has been implemented in Delphi language and run during a predefined CPU time which is fixed to 30, 60, and 120 s for the instances of 50, 100, and 200 jobs, respectively. The machine used for the tests is a Pentium IV PC/AT computer running at 3.2 GHz with 2 GBytes of RAM memory. The performance measure and response variable of the experimental design are the so-called relative percentage deviation (*RPD*) over the optimum or best known solution (upper bound) for each tested instance:

$$\text{Relative percentage deviation } (RPD) = \frac{Heu_{\text{sol}} - Best_{\text{sol}}}{Best_{\text{sol}}} \times 100 \tag{2}$$

where Heu_{sol} is the solution given by any of the replicates of any iterated greedy variation for a given instance and $Best_{\text{sol}}$ is the optimum solution or the lowest known upper bound for that specific Taillard's instance as of November 2015. The results are analyzed by means of the parametric ANOVA technique. Notice that in this case, the three assumptions of this parametric test (normality, homoscedasticity, and independence of the residual) have to be satisfied. In our experiment, the factor *LS* results to be extremely significant, and it creates large differences in the response variable. As expected, applying the local search results in a statistically significant difference in the average *RPD* of 1.46 considering all results, compared with the average *RPD* of 3.73 without local search. Such a large difference generates normality problems. In order to avoid this situation, we study two separate ANOVAs, one for each level of the *LS* factor.

Results of the Simple Iterated Greedy Without Local Search

All remaining factors after fixing the local search have p-values very close to zero in the resulting ANOVA table. As a result, we focus on the F-Ratio, which is the ratio between the variance generated by a given factor and the residual variance in the studied two-level interaction linear model. The higher this ratio, the more significant the factor or interaction is. Figure 4 shows the means plots for the remaining six factors in order of importance.

From Fig. 4 we can observe how all six studied factors have levels and variants that result in statistically significant differences. Observed differences in the average performance of the iterated greedy when a factor has been set to a given level or variant are depicted with nonoverlapping confidence intervals in the plots. The importance of the F-Ratio is shown in Fig. 4 with the most significant factors to the

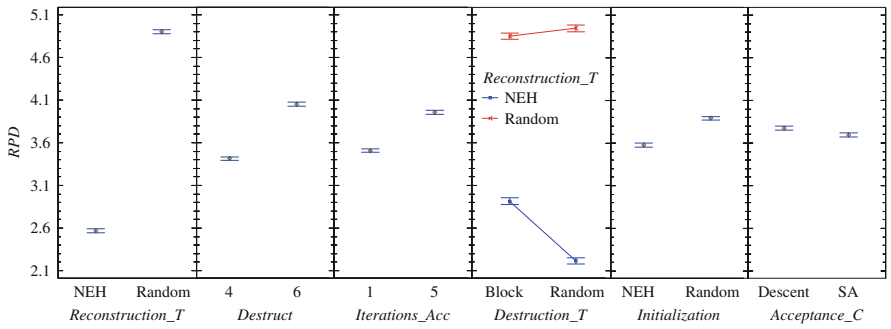

Fig. 4 Means plot of the average relative percentage deviation (*RPD*) and 99% Tukey HSD confidence intervals for the studied factors. Simple iterated greedy without local search

left of the figure and the least significant to the right. As can be seen, without local search, it is clearly preferable to greedily reconstruct the solution as the difference in performance is extremely large. This is one of the main keys behind the iterated greedy algorithm. A random reconstruction of jobs means that first the jobs are randomly extracted and then randomly inserted. This is more or less a sequence of insertion moves. Too many moves (and no local search) result in an algorithm that is not performing well.

The second factor in importance is *Destruct* or d, the number of jobs to be removed in the destruction phase. Following the previous discussion, six jobs impose a rather large overall disruption, and removing only four jobs gives much better results. Notice that a similar conclusion was reached by [92] in their calibration of the iterated greedy for the PFSP. Although not shown here, removing four jobs is better for all the studied instances with 50, 100, and 200 jobs.

The third most important factor results to be *Iterations_Acc*. Among the two tested levels, applying the acceptance criterion at every iteration gives substantially better results. The analysis is continued until all factors have been fixed through the most important effects in the response variable. For example, the third most significant effect in the ANOVA is not a single factor but the interaction between factors *Destruction_T* and *Reconstruction_T*. It is shown in the fourth plot from the left in Fig. 4. We see that both factors interact greatly. When *Reconstruction_T* is set to random, the block destruction appears to be slightly better. However, when *Reconstruction_T* is set to NEH, as in the original NEH method, the destruction has to be done randomly as much better results are obtained. *Initialization* is best set at NEH, but we observe that the difference in performance with the random initialization is rather small. Finally, the acceptance criterion factor (*Acceptance_C*) is also slightly better at SA, but the difference with descent is very small, even though it is statistically significant. More specifically, the average *RPD* given by the simple iterated greedy across all factors with a SA-like acceptance criterion is 3.69%, whereas for the descent criterion it is 3.77%. It is interesting to note that most results confirm the choices made in previous studies as in [92].

Results of the Simple Iterated Greedy with a Local Search Step

Now we proceed with the analysis of the experimental data after focusing on the results of the simple iterated greedy with the local search enabled. In this experiment, the first interesting result is that the observed differences as regards *RPD* among the levels of the factors are much smaller than in the previous one. It is safe to say that the local search is capturing much of the variability of the experiment. Figure 5 contains the means plots for the six factors, in order of importance according to their F-Ratios.

As can be seen, the relative importance of the studied factor is not the same when compared to the previous experiment without local search. The most important factor is now *Iterations_Acc*. Similar to the previous experiment, applying the acceptance criterion at each iteration improves results significantly. The next factor in importance is *Reconstruction_T*. The plot is similar to the previous experiment

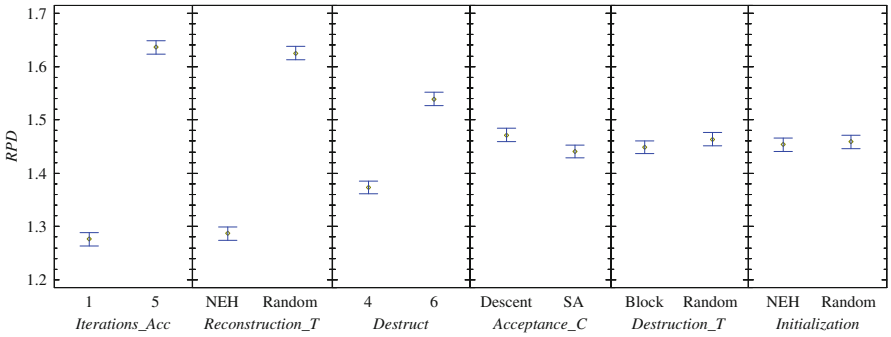

Fig. 5 Means plot of the average relative percentage deviation (*RPD*) and 99% Tukey HSD confidence intervals for the studied factors. Simple iterated greedy with local search enabled

and represents an important result. Considering that the algorithm already carries out a local search step after reconstruction, it could be argued that a random reconstruction might suffice. However, this is not the case. Carrying out a local search step from a randomly reconstructed solution yields, on average, much worse results than if the local search is carried out on a greedily (and therefore, of higher quality) reconstructed solution. This supports the point that iterating over greedy heuristics gives very good results. The remaining factors per importance are *Destruct* and *Acceptance_C*. The last two remaining factors, *Destruction_T* and *Initialization*, are not statistically significant at a 99% confidence level as it can be seen by the fact that their corresponding means plots overlap. Hence, among the two tested types of destruction, it is not important how the jobs are removed. This is probably due to the fact that even though jobs are removed in blocks, they are reinserted one by one. As for the initialization, the advantage gained by an NEH initialization is nullified during the run of the iterated greedy algorithm, confirming the observation of [81].

As a conclusion from this study, the most significant factor is the local search and the NEH reconstruction. Most other factors have less importance. Iterated greedy can even do without a full NEH initialization for the hardest instances of Taillard in the PFSP problem.

IG Applications: Historical Development

Among the examples of iterated greedy algorithms in section "Some Simple Examples of Iterated Greedy Algorithms," we had described an algorithm for the SCP by [50] who have actually called their algorithm a *simulated annealing* algorithm. This naming is probably due to the use of the Metropolis condition as an acceptance criterion, which was also used in the first proposals of simulated annealing [56]. This algorithm has later been extended by the same authors [14]. Somehow related is the more complex algorithm by Marchiori and Steenbeek [62] for the SCP; in

this algorithm a construction starts from a partial solution following the steps of the iterated greedy method; however, instead of obtaining the partial solution by choosing the subsets to be removed, they use a mechanism to choose the subsets of the best solution found so far that are maintained in the partial solution. Maybe surprisingly, in that paper the proposed algorithm is viewed as an evolutionary algorithm. This algorithm is preceded by another algorithm for the unweighted SCP [61], which more directly follows the ideas of the iterated greedy method.

Interestingly, the abovementioned algorithms for the SCP are not the only ones that are directly based on the same destruction–construction process that is the basis of iterated greedy algorithms. In fact, a large number of other algorithms are based on the same principle. However, different names have been used for the underlying method or no specific name at all, which complicates the transfer of knowledge to other researchers.

Probably the earliest techniques that use some mechanism akin to the destructive–constructive moves of iterated greedy are found in the VLSI design automation community and, in particular, in the routing step, where components are connected by wires trying to obey design rules for integrated circuits. In the rip-up and reroute approach [20, 90], some routes related to bottlenecks are removed, and components are reconnected in a different order. The rip-up and reroute technique is commonly used in the design automation community, but the principle underlying this method seems not to have been generalized to the level of metaheuristic for tackling a wide variety of other optimization problems.

The possible use of destructive–constructive moves has also been mentioned earlier in the context of the type of techniques that have become known as strategic oscillation [38–40]. These types of approaches are often embedded into algorithms that make use of tabu search features to provide additional guidance to the search. Some examples of such implementations that have also a clear component related to the iterated greedy methodology are [42, 43, 59].

Among the first researchers to identify the potential of the principles underlying iterated greedy and to formulate these as a general-purpose SLS method are [95]. They called their method *ruin-and-recreate*, where the *ruin* stands for the solution destruction and the *recreate* for the reconstruction of a complete solution.[1] In their seminal article, they applied ruin-and-recreate to the symmetric TSP, the vehicle routing problem with time windows, and to a network design problem, reporting good overall performance of the method. Following this work, a number of other papers have been published using the name ruin-and-recreate; however, not all papers that use the name ruin-and-recreate should be considered as iterated greedy algorithms; for example, [65, 66] essentially uses a random mutation of a current solution instead of a clearly separable destruction phase and greedy solution reconstruction.

[1] Ruin-and-recreate is protected by US patent *Optimization with ruin recreate* No. 6418398; see http://www.patentstorm.us/patents/6418398-fulltext.html.

Ahmadi and Osman [3] study the capacitated clustering $p-$median problem. Among different proposed methods and heuristics, the authors introduce a *periodic construction–deconstruction* procedure, which is basically a simple form of iterated greedy. After a given number of clusters have been built in the constructive method, some of them are randomly deconstructed, and the constructive method is reapplied.

Richmond and Beasley [86] have presented an *iterative construction heuristic* for a problem arising in ore selection. In this problem, processing options needs to be chosen for a number of mining blocks. The destruction operator deletes the processing option for some mining blocks; the size of the destruction is chosen randomly following a uniform distribution in some interval. After the reconstruction of a complete solution, the algorithm accepts only better quality solutions as new incumbent ones. It should be clear from the description that the algorithm follows directly the very same principles of iterated greedy.

Cesta et al. [17] describe an algorithm for a capacitated scheduling problem that makes usage of the ideas underlying the concept of iterated greedy algorithms. They have developed a method that uses a greedy constructive heuristic to generate feasible schedules by introducing precedence constraints. In the destruction phase (called retraction in the chapter), they remove some of the introduced precedence relations to obtain a partial solution and then reconstruct a complete feasible one. They call their method *iterative flattening*, where flattening refers to the reduction of resource conflicts, which is an effect of the introduced precedence constraints. The term iterative flattening has gained some popularity, and several follow-up articles have been published on improvements of the algorithm [63], studies of combinations of basic algorithm components [69], or applications to other problems [70, 71].

The authors of iterative flattening characterize the method in their initial paper as a local search method. Few years before, [96] has proposed the *large neighborhood search* method. The initial proposal consists in the removal of solution components and the reinsertion of solution components by using constraint programming techniques that exploit a tree search and constraint propagation techniques. Even though the method was proposed in a constraint programming framework, where a tree search is used to restore a complete solution instead of simple constructive heuristic, the latter is mentioned as one possible option. In fact, in many later articles on large neighborhood search, simple constructive heuristics are used [78], implementing, hence, directly an iterated greedy method.

Finally, the name *iterated greedy* has been used by [92] and also in [48] to denominate the type of SLS method we describe in this chapter. However, these are not the first publications that are using the name *iterated greedy*. A much earlier mentioning of the term iterated greedy is by Culberson [19]. His iterated greedy algorithm for graph coloring uses a greedy algorithm for generating a coloring, removing the color of all nodes and then reapplying the greedy algorithm using a specific ordering of the nodes that guarantees the next generated coloring to be not worse than the previous one but potentially better. This algorithm may be seen as an extreme case of iterated greedy where the destruction operator destroys the complete solution. However, our view is rather that it is a specific algorithm that works in the way it is proposed mainly for graph coloring.

Relationship to Other Approaches

Iterated greedy is a general-purpose SLS method that has many links to other SLS methods. In the following, we discuss similarities and differences to a number of SLS methods considering other constructive methods, local search methods, and tree search algorithms. These relationships also open up many possibilities for combining techniques proposed for different methods.

Repeated (Greedy) Construction Algorithms

Iterated greedy has natural connections to other methods that repeatedly construct complete candidate solutions. A key difference between iterated greedy and few other repeated construction methods is that the solution construction in iterated greedy usually starts from a (nonempty) partial solution, while the other methods repeatedly generate new candidate solutions starting from *empty* candidate solutions.

A well-known simple such method is the greedy randomized adaptive search procedures (GRASP) that combines a greedily biased but randomized solution construction with a subsequent improvement of the generated candidate solutions by a local search procedure [30, 84]. GRASP in turn extends on some initial ideas of how to generate different solutions by a randomized greedy heuristic [45]. While GRASP does not start the construction from partial solutions, it is an example of a (simple) randomization scheme during the solution construction, which could be included directly in iterated greedy algorithms. An appealing feature of GRASP is the usage of adaptive constructive procedures, and the large number of articles on GRASP [84] may be a source for randomized greedy heuristics to be used in iterated greedy algorithms.

Many useful ideas that underlie other constructive methods may be adopted into iterated greedy algorithms. One such possibility is ways of how to improve the constructive mechanisms, for example, through look-ahead methods or, when taking look-ahead to an extreme, as advocated in the rollout [11] and pilot method [27]. In the rollout and pilot methods for each constructive decision to be taken, a full solution is constructed (or at least well approximated), and the construction decision that results in the best complete solution is taken as the next one. This process is repeated for each construction decision, resulting in a relatively time-consuming constructive approach. However, starting the rollout/pilot method from partial solutions with the resulting reduction in computation time may make this approach promising inside an iterated greedy algorithm.

Squeaky wheel optimization (SWO) [52] uses the idea of biasing the solution construction by information that was gained by analyzing complete candidate solutions. This is done by assigning priorities to specific solution components. The solution reconstruction in SWO is done, differently from iterated greedy, from empty initial solutions corresponding to a complete destruction of the current candidate solution. Within the SWO framework, Aicklin et al. [5] proposed the idea of seeding the reconstruction of a full candidate solution by some partial solution

that was obtained by removing some of the solution components of a complete candidate solution, making it a variant of iterated greedy.

A rather different way of biasing the solution construction is underlying the ant colony optimization (ACO) metaheuristic [23, 24]. In ACO, (artificial) ants implement stochastic solution construction heuristics that make their constructive decisions based on so-called pheromone information and heuristic information associated to specific solution components. The pheromone information tries to bias solution construction toward the best solutions that have been found so far. Again, differently from iterated greedy algorithms, the ants start their solution construction from empty candidate solutions. However, there have been a number of proposals in the ACO area that suggest starting the solution construction from partial solutions that are either stored [1, 2] or obtained by deconstructing complete solutions [103, 104, 109] in a way akin to iterated greedy algorithms. These approaches generally try to transfer the advantages of iterated greedy to ACO algorithms, and for several of these approaches, positive results are reported.

(Perturbative) Local Search Techniques

Iterated greedy has tight links to local search algorithms and, in particular, to very large-scale neighborhood searches [4]. In fact, as also mentioned in section "IG Applications: Historical Development," one of the proposals of iterated greedy style algorithms called the method large neighborhood search Shaw [78]. The analogy stems from the fact that the removal of k solution components in the destruction step and the subsequent reconstruction of a complete candidate solution can be seen as a move in a large neighborhood that is implicitly defined by the number of components to be removed and/or added.[2] The (greedy) reconstruction of a complete candidate solution typical for iterated greedy algorithms can then be seen as a heuristic examination of the neighborhood either in a fully greedy way or using randomization steps–details that simply depend on the particular design and implementation of the construction step in an iterated greedy algorithm. By varying the value of k, the size of the implicitly defined neighborhood is varied. Such changes may be done either (i) randomly within some limited range $[k_{\min}, k_{\max}]$ resembling known strategies in local search methods such as robust tabu search [98], where one tries to avoid overcommitment to a single value for a parameter, (ii) in a more systematic way such as advocated in the strategies defined for variable neighborhood search [44], or (iii) by using feedback from the search performance such as advocated in reactive search methods [9]. An interesting possibility for exploring the neighborhoods is used in the adaptive large neighborhood algorithms

[2]Note that the number of solution components removed in a destruction step may be different from the number of solution components added in the construction step and so we refrain from talking of k-exchange neighborhoods here. A common example where this happens is subset problems such as the SCP we discussed earlier.

by [77, 89], who propose to consider different heuristics to be used in the solution reconstruction. In their method, they additionally adapt the probability with which specific heuristics are chosen over the run-time of the algorithm based on the feedback of the search process.

An alternative to the heuristic exploration of neighborhoods provides methods that try to determine the best possible solution that can be reached from the current one, akin to best-improvement neighborhoods. In the case of large neighborhood searches, this corresponds to an exact examination of all the possible solutions that can be reached from a current partial candidate solution [4, 28]. The original proposal of large neighborhood search by Shaw [96] actually considered such an exact exploration of the resulting neighborhood by means of an constraint programming approach embedded within a branch-and-bound scheme. Due to the large variation in computation time that such a scheme however incurs, alternatives have been tested such as pure insertion-based reconstruction or limited enumeration schemes.

If we consider other local search-based SLS methods, the closest related one is certainly iterated local search [83], especially when iterated greedy algorithms exploit additional (perturbative) local search methods to improve candidate solutions, as outlined in Fig. 3. In that case, the destruction–construction cycle corresponds to what in ILS terms would be a solution perturbation. In general, it seems to be commendable in ILS to apply problem-specific perturbations whenever possible, and the destruction–construction cycle in iterated greedy introduces such problem-specific information through the use of heuristic information. Basic variable neighborhood search is an ILS-type algorithm that systematically varies the strength of the perturbation, and thus the link to iterated greedy is also immediate.

Besides the fact that complete solutions may be improved by an additional local search phase in iterated greedy algorithms, it could be worthwhile to also consider the local re-optimization of the partial solutions that are obtained during the destruction and reconstruction process. It is noteworthy that such occasional local re-optimization of partial candidate solutions has shown to be useful in a number of other contexts such as vehicle routing [16].

Tree Search Algorithms

Constructive mechanisms can be extended to an exhaustive search method by adding a simple backtracking mechanism; adding further bounding schemes, one quickly comes to methods such as branch-and-bound or, more in general, tree search techniques [48]. Hence, it is clear that iterated greedy algorithms also share some relationships to such methods.

Some links have already been mentioned in the previous section, when exact methods are exploited to generate the best possible completion of a partial candidate solution. Such a completion may be generated using tree search methods, and various of these ideas have been explored [96]. Even if tree search is used, the search need not be necessarily complete. For example, Shaw [96] has considered the usage of limited discrepancy search [46], which consists in examining a limited number

of alternative choices in the decision points during the tree generation. Any other variants such as depth-bounded discrepancy search [108] could also be interesting. Another alternative that does not seem to have been applied so far would be to apply beam search [72].

Yet another connection to tree search techniques could rely on the exploitation of lower bound information to generate heuristic information or to prune choices that are guaranteed not to lead to improved solutions. So far, we are not aware of an exploitation of such ideas inside iterated greedy algorithms, although in other constructive SLS algorithms, such uses have been explored [12, 60].

Applications

In this section, we give a short overview of the applications of iterated greedy algorithms. Few applications have already been mentioned in section "IG Applications: Historical Development" when discussing other methods that use the same principle as iterated greedy. Here, we focus mainly on applications that identified their algorithm as an iterated greedy algorithm. In fact, depending on the class of problems for which the respective methods have been proposed, the usage of names to refer to iterated greedy-type methods differs. For example, in the scheduling area, the excellent results of the iterated greedy algorithms for the permutation flow shop scheduling problem have spawned a lot of follow-up work on similar problems, while the name large neighborhood search is frequently found in application to vehicle routing problems, given the prominent role these algorithms have played in that domain.

Iterated Greedy for Scheduling Problems

As mentioned, iterated greedy, as defined in this chapter, was initially applied to the permutation flow shop scheduling problem by [92], so many applications to other scheduling problems were published after that.[3] [93] extended their iterated greedy algorithm to tackle the permutation flow shop scheduling problem when considering additional sequence-dependent setup times and other objectives than makespan, namely, the total tardiness. Other variants of flow shop problems have been studied in [85, 101] (blocking), [75] and [22] (no-wait), [94, 100], and [74] who studied no-idle and mixed no-idle problems. Non-permutation flow shops were approached by [111] or [10]. Distributed flow shop scheduling problems have been solved with iterated greedy algorithms in [57] and [33]. Iterated greedy has also proven valuable in other optimization criteria apart from makespan. Tardiness is studied in [34] or total flowtime in [73] to name just a few. Multi-objective

[3] Various applications of iterative flattening to scheduling problems have been referenced in section "IG Applications: Historical Development".

extensions of iterated greedy have proven effective for Pareto flow shops without and with setups in [64] and in [18], respectively; [25] have embedded an iterated greedy algorithm into the two-phase local search framework to tackle various bi-objective flow shop problems. Parallel machine problems were successfully tackled with iterated greedy algorithms [6, 29, 88]. Iterated greedy algorithms are effective for various single-machine scheduling problems as shown in [21, 112]. Some other more complex problems have been studied, including job-shop scheduling with blocking constraints [80], job-shop scheduling with sequence-dependent setup times and job families [55], hybrid flow shops [110], and real-life problems [105–107].

Iterated Greedy for Routing Problems

As mentioned above, in the context of vehicle routing problems, several algorithms that follow the main structure of an iterated greedy algorithm have been applied but usually using branding under the name large neighborhood search. The article making these approaches popular has already been mentioned [96]. A major impact in that line of applications had the adaptive large neighborhood search approaches [77,89], which led to a large number of follow-up work mainly in the vehicle routing domain. Some overview is also given in [78]. In fact, the usage of such iterated greedy-type methods as local searches inside SLS algorithms for vehicle routing problems is increasingly widespread and can by now be considered a standard in this area. Apart from vehicle routing, also iterated greedy approaches have been proposed to few other routing-type problems such as scheduling and routing problems of freight trains [113] or in the context of TSP variants [54].

Iterated Greedy for Other Problems

There have been a number of other applications where explicitly iterated greedy algorithms have been devised as the main solution techniques. Lozano et al. [58] presented an iterated greedy approach to the maximum diversity problem, where from a set of elements a subset with maximum diversity has to be chosen. After proper tuning, the algorithm was shown to perform better than various competing algorithms and was established as a new state-of-the-art algorithm. García-Martínez et al. [37] have developed an iterated greedy algorithm enhanced by a short tabu list in the destructive phase for the quadratic multiple knapsack problem. Lozano et al. [59] have developed also another iterated greedy algorithm for the quadratic minimum spanning tree problem obtaining excellent results on large instances; further embedding the algorithm into a strategic oscillation approach by essentially extending it with tabu criteria led to further improvements on some problem instance classes. Early implementations of similar ideas for binary quadratic programming have been presented before [43]; more recently, Toyama et al. [102] apply iterated greedy to the same problem tackling also very large-scale instances successfully. Kang et al. [53] study the problem of allocating parallel tasks to processors

in computing systems that are distributed and heterogeneous. Population-based iterated greedy algorithms and iterated greedy algorithms that exploit further exact solutions to large neighborhood searches have been proposed for the maximal covering location problem [87], the goal of which is to cover clients such that the largest amount of demand possible is satisfied. A problem in market segmentation, where a company asks to partition a set of customers subject to some specific requirements related to homogeneity of customers and compactness of the areas is tackled by Huerta-Muñoz et al. [49]. A population-based iterated greedy algorithm has also been applied for delimiting and zoning rural settlements [79] and to the minimum weight vertex cover problem [13]. First applications of iterated greedy algorithms for machine learning tasks, in particular the generation of classification rules, have been explored [76].

Given the wide applicability, the flexibility, and the often high performance of iterated greedy, we would expect this list of applications to further grow significantly in the future.

Conclusions

The main principle of iterated greedy is to build a sequence of solutions by iterating over constructive algorithms through a loop of solution destructions and (re-)constructions. Deconstruction removes solution components resulting in partial solutions from which again full solutions are reconstructed. This loop may be extended by an additional local search phase, where the generated complete candidate solutions are further improved, and by many other techniques from other heuristic and exact search techniques, making iterated greedy a very flexible and malleable method.

We prefer to call this principle iterated greedy because (i) this name directly refers to what the principle implies, namely, making iterative use of construction methods, (ii) it does not obfuscate the name with natural or unnatural analogies, (iii) it is a short and punchy name describing the essence of the method, and (iv) and it has by now been used in a large number of publications. A bit unfortunate is that in the literature there have been proposed several methods under different names that make use of the same (or a very similar) principle including large neighborhood search [78,96], simulated annealing [50], evolutionary heuristic [62], ruin-and-recreate [95], iterative construction search [86], or iterative flattening [17]. The multitude of different names for the same kind of approach can probably be explained in two ways. First, different researchers have a different perspective on the same method, and, thus, the iterated greedy principle can be viewed and proposed, for example, from the perspective of a (perturbative) neighborhood search leading to the heuristic or exact exploration of large neighborhoods or from the perspective of a (constructive) algorithm background where the method is simply iterating through applications of constructive algorithms. Second, the method has been proposed in somewhat different communities such as ruin-and-recreate in a physics journal [95], large neighborhood search at a constraint programming conference [96], and

iterative flattening at an artificial intelligence conference [17], and publications using other names have appeared in operations research journals [50, 86, 92]. It apparently has taken a significant amount of time that these ideas transpired from one community to another.

Even if this leads to some confusion and we maybe contribute to this confusion, we think that what is really important is (i) the principle underlying these methods, (ii) the fact that the iterated greedy principle can lead to very powerful algorithms, and (iii) the fact that iterated greedy is a very flexible method that can easily be combined with other techniques. By making the relationship among the different proposals clear, we hope to contribute also to a transfer of experience between these different algorithms. In any case, we hope that this review chapter will be useful for other researchers in stochastic local search methods by clearly identifying the potential of iterated greedy and in this way also contribute to its further development.

Acknowledgments This work received support from the COMEX project (P7/36) within the Interuniversity Attraction Poles Programme of the Belgian Science Policy Office. Thomas Stützle acknowledges support from the Belgian F.R.S.-FNRS, of which he is a research director. Rubén Ruiz is partially supported by the Spanish Ministry of Economy and Competitiveness, under the project "SCHEYARD – Optimization of Scheduling Problems in Container Yards" (No. DPI2015-65895-R) financed by FEDER funds.

References

1. Acan A (2004) An external memory implementation in ant colony optimization. In: Dorigo M et al (eds) Ant colony optimization and swarm intelligence, 4th international workshop (ANTS 2004). Lecture notes in computer science, vol 3172. Springer, Heidelberg, pp 73–84
2. Acan A (2005) An external partial permutations memory for ant colony optimization. In: Raidl GR, Gottlieb J (eds) Proceedings of EvoCOP 2005 – 5th European conference on evolutionary computation in combinatorial optimization. Lecture notes in computer science, vol 3448. Springer, Heidelberg, pp 1–11
3. Ahmadi S, Osman IH (2004) Density based problem space search for the capacitated clustering p-median problem. Ann Oper Res 131:21–43
4. Ahuja RK, Ergun O, Orlin JB, Punnen AP (2002) A survey of very large-scale neighborhood search techniques. Discret Appl Math 123(1–3):75–102
5. Aickelin U, Burke EK, Li J (2006) Improved squeaky wheel optimisation for driver scheduling. In: Runarsson TP, Beyer HG, Burke EK, Merelo JJ, Whitley LD, Yao X (eds) (2006) Proceedings of PPSN-IX, ninth international conference on parallel problem solving from nature. Lecture notes in computer science, vol 4193. Springer, Heidelberg
6. Arroyo J, Leung JT (2017) An effective iterated greedy algorithm for scheduling unrelated parallel batch machines with non-identical capacities and unequal ready times. Comput Ind Eng 105:84–100
7. Balas E, Ho A (1980) Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. Math Program Study 12:37–60
8. Bartz-Beielstein T, Chiarandini M, Paquete L, Preuss M (eds) (2010) Experimental methods for the analysis of optimization algorithms. Springer, Berlin

9. Battiti R, Brunato M, Mascia F (2008) Reactive search and intelligent optimization. Operations research/computer science interfaces, vol 45. Springer, New York.
10. Benavides AJ, Ritt M (2015) Two simple and effective heuristics for minimizing the makespan in non-permutation flow shops. Comput Oper Res 66:160–169
11. Bertsekas DP, Tsitsiklis JN, Wu C (1997) Rollout algorithms for combinatorial optimization. J Heuristics 3(3):245–262
12. Blum C (2005) Beam-ACO—hybridizing ant colony optimization with beam search: an application to open shop scheduling. Comput Oper Res 32(6):1565–1591
13. Bouamama S, Blum C, Boukerram A (2012) A population-based iterated greedy algorithm for the minimum weight vertex cover problem. Appl Soft Comput 12(6):1632–1639
14. Brusco MJ, Jacobs LW, Thompson GM (1999) A morphing procedure to supplement a simulated annealing heuristic for cost- and coverage-correlated set covering problems. Ann Oper Res 86:611–627
15. Burke EK, Gendreau M, Hyde MR, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: a survey of the state of the art. J Oper Res Soc 64(12):1695–1724
16. Caseau Y, Laburthe F (1999) Heuristics for large constrained vehicle routing problems. J Heuristics 5(3):281–303
17. Cesta A, Oddi A, Smith SF (2000) Iterative flattening: a scalable method for solving multi-capacity scheduling problems. In: Proceedings of AAAI 2000 – seventeenth national conference on artificial intelligence. AAAI Press/MIT Press, Menlo Park, pp 742–747
18. Ciavotta M, Minella G, Ruiz R (2013) Multi-objective sequence dependent setup times flowshop scheduling: a new algorithm and a comprehensive study. Eur J Oper Res 227(2):301–313
19. Culberson JC (1992) Iterated greedy graph coloring and the difficulty landscape. Tech. Rep. 92-07, Department of Computing Science, The University of Alberta, Edmonton, Alberta
20. Dees WA Jr, Karger PG (1982) Automated rip-up and reroute techniques. In: Proceedings of the 19th design automation workshop (DAC'82). IEEE Press, pp 432–439
21. Deng G, Gu X (2014) An iterated greedy algorithm for the single-machine total weighted tardiness problem with sequence-dependent setup times. Int J Syst Sci 45(3):351–362
22. Ding JY, Song S, Gupta JND, Zhang R, Chiong R, Wu C (2015) An improved iterated greedy algorithm with a tabu-based reconstruction strategy for the no-wait flowshop scheduling problem. Appl Soft Comput 30:604–613
23. Dorigo M, Stützle T (2004) Ant colony optimization. MIT Press, Cambridge
24. Dorigo M, Birattari M, Stützle T (2006) Ant colony optimization: artificial ants as a computational intelligence technique. IEEE Comput Intell Mag 1(4):28–39
25. Dubois-Lacoste J, López-Ibáñez M, Stützle T (2011) A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. Comput Oper Res 38(8):1219–1236.
26. Dubois-Lacoste J, Pagnozzi F, Stützle T (2017) An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. Comput Oper Res 81:160–166
27. Duin C, Voß S (1999) The pilot method: a strategy for heuristic repetition with application to the Steiner problem in graphs. Networks 34(3):181–191
28. Dumitrescu I, Stützle T (2009) Usage of exact algorithms to enhance stochastic local search algorithms. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics—hybridizing metaheuristics and mathematical programming. Annals of information systems, vol 10. Springer, New York, pp 103–134
29. Fanjul-Peyro L, Ruiz R (2010) Iterated greedy local search methods for unrelated parallel machine scheduling. Eur J Oper Res 207(1):55–69
30. Feo TA, Resende MGC (1989) A probabilistic heuristic for a computationally difficult set covering problem. Oper Res Lett 8(2):67–71
31. Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. J Glob Optim 6:109–113
32. Fernandez-Viagas V, Framiñán JM (2014) On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. Comput Oper Res 45:60–67

33. Fernandez-Viagas V, Framinan JM (2015) A bounded-search iterated greedy algorithm for the distributed permutation flowshop scheduling problem. Int J Prod Res 53(4):1111–1123
34. Framinan JM, Leisten R (2008) Total tardiness minimization in permutation flow shops: a simple approach based on a variable greedy algorithm. Int J Prod Res 46(22):6479–6498
35. Framiñán JM, Gupta JN, Leisten R (2004) A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. J Oper Res Soc 55(12): 1243–1255
36. Framiñán JM, Leisten R, Ruiz R (2014) Manufacturing scheduling systems: an integrated view on models, methods, and tools. Springer, New York
37. García-Martínez C, Rodriguez FJ, Lozano M (2014) Tabu-enhanced iterated greedy algorithm: a case study in the quadratic multiple knapsack problem. Eur J Oper Res 232(3): 454–463
38. Glover F (1977) Heuristics for integer programming using surrogate constraints. Decis Sci 8:156–166
39. Glover F (1986) Future paths for integer programming and links to artificial intelligence. Comput Oper Res 13(5):533–549
40. Glover F (1989) Tabu search – part I. INFORMS J Comput 1(3):190–206. https://doi.org/10.1287/ijoc.1.3.190
41. Glover F, Kochenberger G (eds) (2002) Handbook of metaheuristics. Kluwer Academic Publishers, Norwell
42. Glover F, Kochenberger GA (1996) Critical even tabu search for multidimensional knapsack problems. In: Osman IH, Kelly JP (eds) Metaheuristics: theory & applications. Kluwer Academic Publishers, Norwell, pp 407–427
43. Glover F, Kochenberger GA, Alidaee B (1998) Adaptive memory tabu search for binary quadratic programs. Manag Sci 44(3):336–345
44. Hansen P, Mladenović N (2001) Variable neighborhood search: principles and applications. Eur J Oper Res 130(3):449–467
45. Hart JP, Shogan AW (1987) Semi-greedy heuristics: an empirical study. Oper Res Lett 6(3):107–114
46. Harvey WD, Ginsberg ML (1995) Limited discrepancy search. In: Mellish CS (ed) Proceedings of the fourteenth international joint conference on artificial intelligence (IJCAI-95). Morgan Kaufmann Publishers, pp 607–615
47. Hejazi SR, Saghafian S (2005) Flowshop-scheduling problems with makespan criterion: a review. Int J Prod Res 43(14):2895–2929
48. Hoos HH, Stützle T (2005) Stochastic local search—foundations and applications. Morgan Kaufmann Publishers, San Francisco
49. Huerta-Muñoz DL, Ríos-Mercado RZ, Ruiz R (2017) An iterated greedy heuristic for a market segmentation problem with multiple attributes. Eur J Oper Res 261(1):75–87
50. Jacobs LW, Brusco MJ (1995) A local search heuristic for large set-covering problems. Nav Res Logist 42(7):1129–1140
51. Johnson DS, McGeoch LA (2002) Experimental analysis of heuristics for the STSP. In: Gutin G, Punnen A (eds) The traveling salesman problem and its variations. Kluwer Academic Publishers, Dordrecht, pp 369–443
52. Joslin DE, Clements DP (1999) Squeaky wheel optimization. J Artif Intell Res 10:353–373
53. Kang Q, He H, Wei J (2013) An effective iterated greedy algorithm for reliability-oriented task allocation in distributed computing systems. J Parallel Distrib Comput 73(8): 1106–1115
54. Karabulut K, Tasgetiren FM (2014) A variable iterated greedy algorithm for the traveling salesman problem with time windows. Inform Sci 279:383–395
55. Kim JS, Park JH, Lee DH (2017) Iterated greedy algorithms to minimize the total family flow time for job-shop scheduling with job families and sequence-dependent set-ups. Eng Optim 49(10):1719–1732
56. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220:671–680

57. Lin SW, Ying KC, Huang CY (2013) Minimising makespan in distributed permutation flowshops using a modified iterated greedy algorithm. Int J Prod Res 51(16):5029–5038
58. Lozano M, Molina D, García-Martínez C (2011) Iterated greedy for the maximum diversity problem. Eur J Oper Res 214(1):31–38
59. Lozano M, Glover F, García-Martínez C, Rodríguez FJ, Martí R (2014) Tabu search with strategic oscillation for the quadratic minimum spanning tree. IIE Trans 46(4): 414–428
60. Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. INFORMS J Comput 11(4):358–369
61. Marchiori E, Steenbeek AG (1998) An iterated heuristic algorithm for the set covering problem. In: Mehlhorn K (ed) Algorithm engineering, 2nd international workshop (WAE'92). Max-Planck-Institut für Informatik, Saarbrücken, pp 155–166
62. Marchiori E, Steenbeek AG (2000) An evolutionary algorithm for large scale set covering problems with application to airline crew scheduling. In: Cagnoni S et al (eds) Real-world applications of evolutionary computing, EvoWorkshops 2000. Lecture notes in computer science, vol 1803. Springer, Heidelberg, pp 367–381
63. Michel LD, van Hentenryck P (2004) Iterative relaxations for iterative flattening in cumulative scheduling. In: Zilberstein S, Koehler J, Koenig S (eds) Proceedings of the fourteenth international conference on automated planning and scheduling (ICAPS 2004). AAAI Press/ MIT Press, Menlo Park, pp 200–208
64. Minella G, Ruiz R, Ciavotta M (2011) Restarted iterated pareto greedy algorithm for multi-objective flowshop scheduling problems. Comput Oper Res 38(11):1521–1533
65. Misevičius A (2003) Genetic algorithm hybridized with ruin and recreate procedure: application to the quadratic assignment problem. Knowl Based Syst 16(5–6):261–268
66. Misevičius A (2003) Ruin and recreate principle based approach for the quadratic assignment problem. In: Cantú-Paz E et al (eds) Genetic and evolutionary computation – GECCO 2003, part I. Lecture notes in computer science, vol 2723. Springer, Heidelberg, pp 598–609
67. Montgomery DC (2012) Design and analysis of experiments, 8th edn. Wiley, New York
68. Nawaz M, Enscore E Jr, Ham I (1983) A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. Omega 11(1):91–95
69. Oddi A, Cesta A, Policella N, Smith SF (2008) Combining variants of iterative flattening search. Eng Appl Artif Intell 21(5):683–690
70. Oddi A, Cesta A, Policella N, Smith SF (2010) Iterative flattening search for resource constrained scheduling. J Intell Manuf 21(1):17–30
71. Oddi A, Rasconi R, Cesta A, Smith SF (2011) Iterative flattening search for the flexible job shop scheduling problem. In: Walsh T (ed) Proceedings of the twenty-second international joint conference on artificial intelligence (IJCAI-11). IJCAI/AAAI Press, Menlo Park, pp 1991–1996
72. Ow PS, Morton TE (1988) Filtered beam search in scheduling. Int J Prod Res 26:297–307
73. Pan QK, Ruiz R (2012) Local search methods for the flowshop scheduling problem with flowtime minimization. Eur J Oper Res 222(1):31–43
74. Pan QK, Ruiz R (2014) An effective iterated greedy algorithm for the mixed no-idle flowshop scheduling problem. Omega 44(1):41–50
75. Pan QK, Wang L, Zhao BH (2008) An improved iterated greedy algorithm for the no-wait flow shop scheduling problem with makespan criterion. Int J Adv Manuf Tech 38(7–8): 778–786
76. Pedraza JA, García-Martínez C, Cano A, Ventura S (2014) Classification rule mining with iterated greedy. In: Polycarpou MM, de Carvalho ACPLF, Pan J, Wozniak M, Quintián H, Corchado E (eds) Hybrid artificial intelligence systems – 9th international conference (HAIS 2014), Salamanca, 11–13 June 2014. Proceedings. Lecture notes in computer science, vol 8480. Springer, Heidelberg, pp 585–596
77. Pisinger D, Ropke S (2007) A general heuristic for vehicle routing problems. Comput Oper Res 34(8):2403–2435

78. Pisinger D, Ropke S (2010) Large neighborhood search. In: Gendreau M, Potvin JY (eds) (2010) Handbook of metaheuristics. International series in operations research & management science, vol 146, 2nd edn. Springer, New York, pp 399–419
79. Porta J, Parapar J, Doallo R, Barbosa V, Santé I, Crecente R, Díaz C (2013) A population-based iterated greedy algorithm for the delimitation and zoning of rural settlements. Comput Environ Urban Syst 39:12–26
80. Pranzo M, Pacciarelli D (2016) An iterated greedy metaheuristic for the blocking job shop scheduling problem. J Heuristics 22(4):587–611.
81. Rad SF, Ruiz R, Boroojerdian N (2009) New high performing heuristics for minimizing makespan in permutation flowshops. Omega 37(2):331–345
82. Ramalhinho Lourenço H, Martin O, Stützle T (2002) Iterated local search. In: Glover F, Kochenberger G (eds) (2002) Handbook of metaheuristics. Kluwer Academic Publishers, Norwell, pp 321–353
83. Ramalhinho Lourenço H, Martin O, Stützle T (2010) Iterated local search: framework and applications. In: Gendreau M, Potvin JY (eds) (2010) Handbook of metaheuristics. International series in operations research & management science, vol 146, 2nd edn. Springer, New York, chap 9, pp 363–397
84. Resende MGC, Ribeiro CC (2010) Greedy randomized adaptive search procedures: advances, hybridizations, and applications. In: Gendreau M, Potvin JY (eds) (2010) Handbook of metaheuristics. International series in operations research & management science, vol 146, 2nd edn. Springer, New York, pp 283–319
85. Ribas I, Companys R, Tort-Martorell X (2011) An iterated greedy algorithm for the flowshop scheduling problem with blocking. Omega 39(3):293–301
86. Richmond AJ, Beasley JE (2004) An iterative construction heuristic for the ore selection problem. J Heuristics 10(2):153–167
87. Rodríguez FJ, Blum C, Lozano M, García-Martínez C (2012) Iterated greedy algorithms for the maximal covering location problem. In: Hao JK, Middendorf M (eds) Proceedings of EvoCOP 2012 – 12th European conference on evolutionary computation in combinatorial optimization. Lecture notes in computer science, vol 7245. Springer, Heidelberg, pp 172–181
88. Rodriguez FJ, Lozano M, Blum C, García-Martínez C (2013) An iterated greedy algorithm for the large-scale unrelated parallel machines scheduling problem. Comput Oper Res 40(7):1829–1841
89. Ropke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problme with time windows. Transp Sci 40(4):455–472
90. Rubin F (1974) An iterative technique for printed wire routing. In: Proceedings of the 11th design automation workshop (DAC'74). IEEE Press, pp 308–313
91. Ruiz R, Maroto C (2005) A comprehensive review and evaluation of permutation flowshop heuristics. Eur J Oper Res 165(2):479–494
92. Ruiz R, Stützle T (2007) A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. Eur J Oper Res 177(3):2033–2049
93. Ruiz R, Stützle T (2008) An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. Eur J Oper Res 187(3):1143–1159
94. Ruiz R, Vallada E, Fernández-Martínez C (2009) Scheduling in flowshops with no-idle machines. In: Chakraborty UK (ed) Computational intelligence in flow shop and job shop scheduling. Studies in computational intelligence, vol 230. Springer, Berlin, pp 21–51
95. Schrimpf G, Schneider J, Stamm-Wilbrandt H, Dueck G (2000) Record breaking optimization results using the ruin and recreate principle. J Comput Phys 159(2):139–171
96. Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Maher M, Puget JF (eds) Principles and practice of constraint programming, CP98. Lecture notes in computer science, vol 1520. Springer, Heidelberg, pp 417–431
97. Taillard ÉD (1990) Some efficient heuristic methods for the flow shop sequencing problem. Eur J Oper Res 47(1):65–74

98. Taillard ÉD (1991) Robust taboo search for the quadratic assignment problem. Parallel Comput 17(4-5):443–455
99. Taillard ÉD (1993) Benchmarks for basic scheduling problems. Eur J Oper Res 64(2): 278–285
100. Tasgetiren FM, Pan QK, Suganthan PN, Buyukdagli O (2013) A variable iterated greedy algorithm with differential evolution for the no-idle permutation flowshop scheduling problem. Comput Oper Res 40(7):1729–1743
101. Tasgetiren MF, Kizilay D, Pan QK, Suganthan PN (2017) Iterated greedy algorithms for the blocking flowshop scheduling problem with makespan criterion. Comput Oper Res 77: 111–126
102. Toyama F, Shoji K, Mori H, Miyamichi J (2012) An iterated greedy algorithm for the binary quadratic programming problem. In: Joint 6th international conference on soft computing and intelligent systems (SCIS) and 13th international symposium on advanced intelligent systems (ISIS), 2012. IEEE Press, pp 2183–2188
103. Tsutsui S (2006) cAS: ant colony optimization with cunning ants. In: Runarsson TP, Beyer HG, Burke EK, Merelo JJ, Whitley LD, Yao X (eds) (2006) Proceedings of PPSN-IX, ninth international conference on parallel problem solving from nature. Lecture notes in computer science, vol 4193. Springer, Heidelberg, pp 162–171
104. Tsutsui S (2007) Ant colony optimization with cunning ants. Trans Jpn Soc Artif Intell 22: 29–36.
105. Urlings T, Ruiz R (2007) Local search in complex scheduling problems. In: Stützle T, Birattari M, Hoos HH (eds) Engineering stochastic local search algorithms. Designing, implementing and analyzing effective heuristics. Lecture notes in computer science, vol 4638. Springer, Brussels, Belgium, pp 202–206
106. Urlings T, Ruiz R, Sivrikaya-Şerifoğlu F (2010) Genetic algorithms for complex hybrid flexible flow line problems. Int J Metaheuristics 1(1):30–54
107. Urlings T, Ruiz R, Stützle T (2010) Shifting representation search for hybrid flexible flowline problems. Eur J Oper Res 207(2):1086–1095.
108. Walsh T (1997) Depth-bounded discrepancy search. In: Pollack ME (ed) Proceedings of the fifteenth international joint conference on artificial intelligence (IJCAI-97). Morgan Kaufmann Publishers, pp 1388–1395
109. Wiesemann W, Stützle T (2006) Iterated ants: an experimental study for the quadratic assignment problem. In: Dorigo M, et al. (eds) Ant colony optimization and swarm intelligence, 5th international workshop, ANTS 2006. Lecture notes in computer science, vol 4150. Springer, Heidelberg, pp 179–190
110. Ying KC (2008) An iterated greedy heuristic for multistage hybrid flowshop scheduling problems with multiprocessor tasks. IEEE Trans Evol Comput 60(6):810–817
111. Ying KC (2008) Solving non-permutation flowshop scheduling problems by an effective iterated greedy heuristic. Int J Adv Manuf Tech 38(3–4):348–354
112. Ying KC, Lin SW, Huang CY (2009) Sequencing single-machine tardiness problems with sequence dependent setup times using an iterated greedy heuristic. Expert Syst Appl 36(3):7087–7092
113. Yuan Z, Fügenschuh A, Homfeld H, Balaprakash P, Stützle T, Schoch M (2008) Iterated greedy algorithms for a real-world cyclic train scheduling problem. In: Blesa MJ, Blum C, Cotta C, Fernández AJ, Gallardo JE, Roli A, Sampels M (eds) Hybrid metaheuristics. Lecture notes in computer science, vol 5296. Springer, Heidelberg, pp 102–116
114. Zilberstein S (1996) Using anytime algorithms in intelligent systems. AI Mag 17(3):73–83