A Study of Greedy, Local Search and Ant Colony Optimization Approaches for Car Sequencing Problems

Jens Gottlieb[1], Markus Puchta[1], and Christine Solnon[2]

[1] SAP AG
Neurottstr. 16, 69190 Walldorf, Germany
jens.gottlieb@sap.com, markus.puchta@sap.com
[2] LIRIS, Nautibus, University Lyon I
43 Bd du 11 novembre, 69 622 Villeurbanne cedex, France
csolnon@bat710.univ-lyon1.fr

Abstract. This paper describes and compares several heuristic approaches for the car sequencing problem. We first study greedy heuristics, and show that dynamic ones clearly outperform their static counterparts. We then describe local search and ant colony optimization (ACO) approaches, that both integrate greedy heuristics, and experimentally compare them on benchmark instances. ACO yields the best solution quality for smaller time limits, and it is comparable to local search for larger limits. Our best algorithms proved one instance being feasible, for which it was formerly unknown whether it is satisfiable or not.

1 Introduction

The car sequencing problem involves scheduling cars along an assembly line, in order to install options (e.g. sun-roof or air-conditioning) on them. Each option is installed by a different station, designed to handle at most a certain percentage of the cars passing along the assembly line, and the cars requiring this option must be spaced such that the capacity of the station is never exceeded.

This problem is NP-complete [5]. It has been formulated as a constraint satisfaction problem (CSP), and is a classical benchmark for constraint solvers [3, 6, 13]. Most of these CSP solvers use a complete tree-search approach to explore the search space in a systematic way, until either a solution is found, or the problem is proven to have no solution. In order to reduce the search space, this approach is combined with filtering techniques that narrow the variables' domains. In particular, a dedicated filtering algorithm has been proposed for handling capacity constraints of the car sequencing problem [9]. This filtering algorithm is very effective to solve some hardly constrained feasible instances, or to prove infeasibility of some over-constrained instances. However, on some other instances, it cannot reduce domains enough to make complete search tractable.

Hence, different incomplete approaches have been proposed, that leave out exhaustivity, trying to quickly find approximately optimal solutions in an opportunistic way, e.g., local search [1, 7, 8], genetic algorithms [14] or ant colony optimization (ACO) approaches [11]. This paper describes and compares three incomplete approaches for the car sequencing problem.

Section 2 introduces the car sequencing problem, for which several greedy heuristics are described and examined in section 3. Section 4 presents a local search approach, adapted from [8], and section 5 describes an ACO approach that is an improved version of [11]. Empirical results and comparisons are given in section 6, followed by the conclusions in section 7.

2 The Car Sequencing Problem

2.1 Formalization

A car sequencing problem is defined by a tuple (C, O, p, q, r), where $C=\{c_1, .., c_n\}$ is the set of cars to be produced and $O=\{o_1, .., o_m\}$ is the set of different options. The two functions $p : O \rightarrow \mathbb{N}$ and $q : O \rightarrow \mathbb{N}$ define the capacity constraint associated with each option $o_i \in O$, i.e. for any sequence of $q(o_i)$ consecutive cars on the line, at most $p(o_i)$ of them may require o_i. The function $r : C \times O \rightarrow \{0, 1\}$ defines options requirements, i.e., for each car $c_i \in C$ and for each option $o_j \in O$, $r(c_i, o_j)$ returns 1 if o_j must be installed on c_i, and 0 otherwise. Note that two cars may require the same configuration of options. To evaluate the *difference* of two cars, we introduce the function $d : C \times C \rightarrow \mathbb{N}$ that returns the number of different options two cars require, i.e., $d(c_i, c_j) = \sum_{o_k \in O} |r(c_i, o_k) - r(c_j, o_k)|$.

Solving a car sequencing problem involves finding an arrangement of the cars in a sequence, defining the order in which they will pass along the assembly line, such that the capacity constraints are met. We shall use the following notations to denote and manipulate sequences:

- a *sequence*, noted $\pi =< c_{i_1}, c_{i_2}, \ldots, c_{i_k} >$, is a succession of cars;
- the *length* of a sequence π, noted $|\pi|$, is the number of cars that it contains;
- the *concatenation* of 2 sequences π_1 and π_2, noted $\pi_1 \cdot \pi_2$, is the sequence composed of the cars of π_1 followed by the cars of π_2;
- a sequence π_f is a *factor* of another sequence π, noted $\pi_f \subseteq \pi$, if there exists two (possibly empty) sequences π_1 and π_2 such that $\pi = \pi_1 \cdot \pi_f \cdot \pi_2$;
- the *number of cars that require an option* o_i within a sequence π (resp. within a set S of cars) is noted $r(\pi, o_i)$ (resp. $r(S, o_i)$) and is defined by $r(\pi, o_i) = \sum_{<c_l> \subseteq \pi} r(c_l, o_i)$ (resp. $r(S, o_i) = \sum_{c_l \in S} r(c_l, o_i)$);
- the *cost* of a sequence π is the number of violated capacity constraints, i.e.,

$$cost(\pi) = \sum_{o_i \in O} \sum_{\substack{\pi_k \subseteq \pi \text{ so that} \\ |\pi_k| = q(o_i)}} violation(\pi_k, o_i)$$

$$\text{where } \; violation(\pi_k, o_i) = \begin{cases} 0 & \text{if } r(\pi_k, o_i) \leq p(o_i); \\ 1 & \text{otherwise.} \end{cases}$$

We can now define the solution process of a car sequencing problem (C, O, p, q, r) as the search of a minimal cost sequence composed of the cars to be produced.

2.2 Utilization Rate

The difficulty of an instance depends on the number of cars to be produced and the number of different options configurations, but also on the utilization rate of the different options [10]. The utilization rate of an option o_i corresponds to the ratio of the number of cars requiring o_i with respect to the maximum number of cars in a sequence which could have o_i while satisfying its capacity constraint, i.e., $utilRate(o_i) = \frac{r(C,o_i)\cdot q(o_i)}{|C|\cdot p(o_i)}$. An utilization rate greater than 1 indicates that the demand is higher than the capacity, so that the capacity of the station will inevitably be exceeded; an utilization rate close to 0 indicates that the demand is very low with respect to the capacity of the station.

2.3 Test Suites

All considered instances are available in the benchmark library CSPLib [6]. The first test suite contains 70 problem instances, used in [1, 7, 11] and grouped into 7 sets of 10 instances per utilization rate 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, and 0.90; e.g. the instances within the group 0.70 are named `70-01`, ..., `70-10`, and we refer to the group as `70-*`. All these instances are feasible ones, and have 200 cars to sequence, 5 options, and from 17 to 30 different options configurations. We also consider a second test suite composed of 9 harder instances, some of which were used in [9]. These instances have 100 cars to sequence, 5 options, from 18 to 24 different configurations, and high utilization rates, equal to 1 for some options; some of them are feasible, whereas some others are over-constrained.

3 Greedy Algorithms

3.1 The Basic Idea and Different Heuristics

Given a car sequencing problem, one can build a sequence in a greedy way, by iteratively choosing the next car to sequence with respect to some given heuristic function to optimize. Obviously, one should choose, at each step, a car that introduces the smallest number of new constraint violations, i.e., given a partial sequence π, the next car is selected within those cars c_j that minimize

$$newViolations(\pi, c_j) = \sum_{o_i \in O} r(c_j, o_i) \cdot violation(lastCars(\pi \cdot < c_j >, q(o_i)), o_i)$$

where $lastCars(\pi', k)$ is the sequence composed of the k last cars of π' if $\mid \pi' \mid \geq k$, otherwise $lastCars(\pi', k) = \pi'$. The set of candidate cars that minimize this *newViolations* function usually contains more than one car so that we may use another heuristic to break further ties. We consider different heuristics:

1. Random choice (Rand): choose randomly a car within the set of candidates.

2. Static Highest Utilization Rates (SHU): the idea, introduced in [10], is to choose first the cars that require the option with highest utilization rate, and to break ties by choosing cars that require the option with second highest utilization rate, and so on. More formally, we choose the car that maximizes the heuristic function $\eta_{\text{SHU}}(c_j) = \sum_{o_i \in O} r(c_j, o_i) \cdot \text{weight}(o_i)$ where $\text{weight}(o_i) = 2^k$ if o_i is the option with the k^{th} smallest utilization rate.
3. Dynamic Highest Utilization Rates (DHU): the heuristic function η_{SHU} is static in the sense that it does not depend on the partial sequence π under construction. We can define a similar function η_{DHU} that is dynamic, by updating utilization rates each time a car is added, i.e., the utilization rate considered to compute the weight of an option o_i is defined by: $\text{dynUtilRate}(o_i, \pi) = \frac{[r(C,o_i) - r(\pi,o_i)] \cdot q(o_i)}{p(o_i) \cdot [|C| - |\pi|]}$.
4. Static Sum of Utilization Rates (SSU): instead of ranking utilization rates, from the highest to the lowest one, we can simply consider the sum of the required options utilization rates. More formally, we choose the car that maximizes the heuristic function $\eta_{\text{SSU}}(c_j) = \sum_{o_i \in O} r(c_j, o_i) \cdot \text{utilRate}(o_i)$.
5. Dynamic Sum of utilization rates (DSU): we can also make the SSU heuristic dynamic by considering the sum of dynamic utilization rates, i.e., $\eta_{\text{DSU}}(c_j, \pi) = \sum_{o_i \in O} r(cj, o_i) \cdot \text{dynUtilRate}(o_i, \pi)$.
6. Dynamic Even Distribution (DED): the four previous heuristics choose cars that require options with high utilization rates, whereas the idea of this heuristic is to favor an even distribution of the options. If the average number of cars that require an option is lower in the sequence under construction π than in C, the set of cars, then we favor these cars, and vice versa. More formally, we choose the car that maximizes the heuristic function

$$\eta_{\text{DED}}(c_j, \pi) = \sum_{o_i \in O} (r(c_j, o_i) = 0) \ \text{xor} \ \left(\frac{r(C, o_i)}{n} > \frac{r(\pi, o_i)}{| \pi |} \right)$$

This heuristic function is not defined for the first car, when $|\pi| = 0$. Therefore, the first car c_j is selected randomly among those cars requiring the maximum number of options.

3.2 Empirical Comparison of Greedy Heuristics

Table 1 displays results obtained on the benchmark instances described in section 2.3. Remark that standard deviations for SHU and DHU are null. Indeed, all runs on a same instance always construct the same sequence as, at each step, there is only one car that maximizes the η function. For SSU and DSU, standard deviations are rather small, and on some instances they are null, as sums of utilization rates often have different values. For DED, standard deviations are higher, as the η_{DED} function can only take $|O|$ different values and in most cases there is more than one car that maximize it.

When comparing the average costs of the sequences built with the different heuristics, we observe the dynamic heuristics DSU and DHU obtaining rather similar results, whereas DSU and DHU always outperform their static counterpart SSU and SHU. DED is better than DSU and DHU on instances with utilization rates lower than 0.90, whereas on the instances `90-*` and the second test suite it is worse. On the one hand, we perceive the instances from suite 1 as easy since each of them could be solved by at least one heuristic (see table 2). On the other hand, all heuristics failed to find a feasible solution for each instance from suite 2. Therefore, suite 2 is considered as more difficult.

Suite	No.	Rand	SSU	DSU	SHU	DHU	DED
1	`60-*`	20.4 (5.1)	10.2 (0.5)	1.6 (0.0)	10.5 (0.0)	1.4 (0.0)	0.8 (0.9)
	`65-*`	17.0 (5.0)	13.9 (0.3)	2.3 (0.0)	16.4 (0.0)	2.1 (0.0)	0.8 (1.0)
	`70-*`	13.8 (4.9)	17.4 (0.4)	3.0 (0.4)	18.6 (0.0)	3.6 (0.0)	0.7 (0.9)
	`75-*`	12.0 (5.0)	18.2 (0.2)	5.6 (0.4)	18.5 (0.0)	6.7 (0.0)	0.9 (1.0)
	`80-*`	17.8 (6.3)	16.6 (1.1)	4.3 (0.6)	17.2 (0.0)	5.0 (0.0)	1.0 (1.1)
	`85-*`	29.4 (7.3)	11.5 (0.6)	4.2 (0.4)	12.4 (0.0)	3.3 (0.0)	1.8 (1.2)
	`90-*`	54.2 (8.3)	5.9 (0.6)	1.2 (0.3)	6.1 (0.0)	1.6 (0.0)	4.3 (1.7)
2	`10-93`	59.1 (6.5)	11.0 (0.0)	10.4 (1.6)	11.0 (0.0)	12.0 (0.0)	14.1 (2.5)
	`16-81`	43.3 (5.1)	9.7 (1.2)	7.7 (0.9)	17.0 (0.0)	11.0 (0.0)	9.5 (2.7)
	`19-71`	56.0 (6.7)	13.6 (1.7)	8.5 (1.1)	13.0 (0.0)	7.0 (0.0)	10.3 (2.4)
	`21-90`	49.1 (5.6)	8.0 (0.0)	5.5 (0.7)	9.0 (0.0)	7.0 (0.0)	9.1 (1.9)
	`26-82`	40.6 (5.9)	6.0 (0.0)	3.6 (0.9)	7.0 (0.0)	3.0 (0.0)	8.8 (2.2)
	`36-92`	50.4 (6.2)	5.0 (0.0)	8.5 (0.5)	8.0 (0.0)	7.0 (0.0)	12.7 (2.3)
	`41-66`	34.1 (5.8)	4.7 (1.4)	3.7 (0.7)	6.0 (0.0)	2.0 (0.0)	6.5 (2.0)
	`4-72`	46.6 (5.8)	4.7 (0.6)	5.0 (0.0)	4.0 (0.0)	2.0 (0.0)	13.3 (3.2)
	`6-76`	28.1 (4.8)	6.0 (0.0)	6.0 (0.0)	6.0 (0.0)	6.0 (0.0)	9.1 (1.4)

Table 1. Results for greedy approaches (average costs on 500 runs, standard deviations in brackets).

No.	Rand	SSU	DSU	SHU	DHU	DED
`60-*`	0.002	0.002	0.400	0.000	0.500	0.547
`65-*`	0.002	0.000	0.100	0.100	0.200	0.536
`70-*`	0.004	0.000	0.000	0.000	0.000	0.580
`75-*`	0.005	0.000	0.200	0.000	0.100	0.507
`80-*`	0.001	0.000	0.116	0.000	0.100	0.487
`85-*`	0.000	0.200	0.218	0.100	0.200	0.251
`90-*`	0.000	0.300	0.500	0.300	0.500	0.114

Table 2. Success rates for greedy approaches and suite 1 (500 runs per instance).

4 Local Search

We consider the local search (LS) procedure proposed in [8] for general car sequencing problems involving several different types of constraints, including the one that is the subject of this paper. First, the initial solution is generated randomly – like in [8] – or by some greedy heuristic described in section 3. This solution is then modified by LS, where each iteration consists of randomly selecting one move type out of six different types. Then, a randomly chosen move of the selected type is evaluated regarding the current solution candidate, and the move is accepted and applied, if it does not deteriorate the solution quality. Otherwise the move is rejected and another one is tried on the current solution. The search process is terminated if a feasible solution is found or the evaluation limit is reached.

Table 3 summarizes the move types in a formal way. Insert moves one car from its current position to another, and Swap exchanges two cars. A special case of Swap is SwapS, which exchanges two cars that are similar but not identical regarding the options they require. SwapT is a transposition, another special case of Swap exchanging two neighbouring cars. Lin2Opt inverts a factor, which is defined by $invert(<>) = <>$ and $invert(\pi\cdot < c_l >) = < c_l > \cdot invert(\pi)$. The last type, Random, randomly shuffles a factor, formalized by $shuffle(<>) = <>$ and $shuffle(\pi_1\cdot < c_l > \cdot\pi_2) = < c_l > \cdot shuffle(\pi_1 \cdot \pi_2)$ with probability $\frac{1}{1+|\pi_1|+|\pi_2|}$.

Besides the specific restrictions listed in table 3, the length of a modified factor is bounded by $n/4$ for all move types, in order to allow fast evaluations. Note that in particular SwapS and SwapT can be evaluated quite quickly since only two neighbouring cars and two options are affected, respectively.

Type	$\pi \rightarrow \pi'$	Restriction
Insert	$\pi_1\cdot < c_l > \cdot\pi_2 \cdot \pi_3 \rightarrow \pi_1 \cdot \pi_2\cdot < c_l > \cdot\pi_3$ (forward) $\pi_1 \cdot \pi_2\cdot < c_l > \cdot\pi_3 \rightarrow \pi_1\cdot < c_l > \cdot\pi_2 \cdot \pi_3$ (backward)	$1 \leq \|\pi_2\|$
Swap	$\pi_1\cdot < c_{l_1} > \cdot\pi_2\cdot < c_{l_2} > \cdot\pi_3 \rightarrow \pi_1\cdot < c_{l_2} > \cdot\pi_2 < c_{l_1} > \cdot\pi_3$	$1 \leq d(c_{l_1}, c_{l_2})$
SwapS	$\pi_1\cdot < c_{l_1} > \cdot\pi_2\cdot < c_{l_2} > \cdot\pi_3 \rightarrow \pi_1\cdot < c_{l_2} > \cdot\pi_2 < c_{l_1} > \cdot\pi_3$	$1 \leq d(c_{l_1}, c_{l_2}) \leq 2$
SwapT	$\pi_1\cdot < c_{l_1} > \cdot < c_{l_2} > \cdot\pi_2 \rightarrow \pi_1\cdot < c_{l_2} > \cdot < c_{l_1} > \cdot\pi_2$	$1 \leq d(c_{l_1}, c_{l_2})$
Lin2Opt	$\pi_1 \cdot \pi_2 \cdot \pi_3 \rightarrow \pi_1 \cdot invert(\pi_2) \cdot \pi_3$	$2 \leq \|\pi_2\|$
Random	$\pi_1 \cdot \pi_2 \cdot \pi_3 \rightarrow \pi_1 \cdot shuffle(\pi_2) \cdot \pi_3$	$2 \leq \|\pi_2\|$

Table 3. Formal description of the move types used within LS.

5 Ant Colony Optimization

The Ant Colony Optimization (ACO) metaheuristic is inspired by the collective behaviour of real ant colonies, and it has been used to solve many hard combinatorial optimization problems [4]. In particular, [11] describes an ACO algorithm for solving permutation CSPs, the goal of which is to find a permutation of n known values, to be assigned to n variables, under some constraints. This algorithm has been designed to solve any permutation CSP in a generic way, and it has been illustrated on different problems. We now describe an improved version of it that is more particularly dedicated to the car sequencing problem. This new algorithm mainly introduces three new features: first, it uses an elitist strategy, so that pheromone is used to break the tie between the “best” cars only; second, it integrates features borrowed from [12] in order to favor exploration; finally, it uses the heuristic functions introduced in section 3 to guide ants.

Our new algorithm follows the classical ACO scheme: first, pheromone trails are initialized; then, at each cycle every ant constructs a sequence, and pheromone trails are updated; the algorithm stops iterating either when an ant has found a solution, or when a maximum number of cycles has been performed.

Construction graph and pheromone trails initialization. To build sequences, ants communicate by laying pheromone on the edges of a complete directed graph which associates a vertex with each car $c_i \in C$. There is an extra vertex, denoted by $c_{nest} \notin C$, from which ants start constructing sequences (this extra vertex will be considered as a car that requires no option). The amount of pheromone on an edge (c_i, c_j) is noted $\tau(c_i, c_j)$ and represents the learnt desirability of sequencing c_j just after c_i.

As proposed in [12], and contrary to our previous ACO algorithm, we explicitly impose lower and upper bounds τ_{min} and τ_{max} on pheromone trails (with $0 < \tau_{min} < \tau_{max}$). The goal is to favor a larger exploration of the search space by preventing the relative differences between pheromone trails from becoming too extreme during processing. Also, pheromone trails are initialized to τ_{max}, thus achieving a higher exploration of the search space during the first cycles.

Construction of sequences by ants. The algorithmic scheme of the construction of a sequence π by an ant is sketched below:

$\pi \leftarrow < c_{nest} >$
while $\mid \pi \mid \leq \mid C \mid$ **do**
  $cand \leftarrow \{c_k \in C{-}\pi \mid \forall c_j \in C{-}\pi,\ ((d(c_k, c_j) = 0) \Rightarrow (k \leq j))$ and $newViolations(\pi, c_k) \leq newViolations(\pi, c_j)\}$
  let c_i be the last car sequenced in π (i.e., $\pi = \pi' \cdot < c_i >$)
  choose $c_j \in cand$ with probability $p_{c_i c_j} = \frac{[\tau(c_i,c_j)]^\alpha [\eta(c_j,\pi)]^\beta}{\sum_{c_k \in cand} [\tau(c_i,c_k)]^\alpha [\eta(c_k,\pi)]^\beta}$
  $\pi \leftarrow \pi \cdot < c_j >$
end while

Remark that, at each iteration, the choice of a car is done within a restricted set of candidates *cand*, instead of the set of all non sequenced cars $C - \pi$: in order to break symmetries, we only consider cars that require different options configurations; in order to introduce an elitist strategy, we select among the best cars with respect to the number of new constraint violations.

The choice of a car c_j in the candidates' set is done with respect to a probability that both depends on a pheromone factor τ, and a heuristic factor η. This heuristic factor can be any of the heuristic functions described in section 3.

Updating pheromone trails. After every ant has constructed a sequence, pheromone trails are updated according to ACO. First, all pheromone trails are decreased in order to simulate evaporation, i.e., for each edge (c_i, c_j), the quantity of pheromone $\tau(c_i, c_j)$ is multiplied by an evaporation parameter ρ such that

$0 \leq \rho \leq 1$. Then, the best ants of the cycle deposit pheromone, i.e., for each sequence π constructed during the cycle, if the cost of π is minimal for this cycle then, for each couple of consecutive cars $< c_j, c_k > \subseteq \pi$, we increment $\tau(c_j, c_k)$ with $1/cost(\pi)$.

6 Experimental comparison of ACO and LS

6.1 Experimental Setup

We consider two ACO variants obtained by setting parameters to different values. The first variant, called the "Ant" variant, uses α=1, ρ=0.99, τ_{min}=0.01 and τ_{max}=4. The second variant, referred to as "Iter", ignores pheromone by using α=0, ρ=1, τ_{min}=τ_{max}=1. This resembles an iterated greedy algorithm that repeatedly produces solutions by a randomized greedy heuristic. For both variants, we set β to 6 and the number of ants to 15, and we examine combinations with the greedy heuristics DHU, DSU and Rand described in section 3.

Regarding LS, two variants are considered that differ in the algorithm producing the initial solution. The first variant uses a pure random sequence, whereas the second uses the greedy heuristic DED from section 3.

We use different evaluation limits for ACO and LS, because e.g. on instance `10-93` of suite 2, 15 000 solutions produced by ACO and 200 000 solutions generated by LS need roughly the same CPU time, 20 seconds on a 300 MHz PC.

6.2 Results for Test Suite 1

Table 4 reports results for the first test suite, limiting the number of cycles of Ant and Iter to 100, and the number of moves of LS to 20 000. Note that within 100 cycles, pheromone cannot influence ants, so that Iter and Ant obtain similar results. Actually, instances of this first suite are rather easy ones, so that they can be very quickly solved by all algorithms, provided they use a good heuristic (i.e., DSU, DHU, or DED).

When no heuristic is used (Rand variants), LS is still able to solve all instances, for almost all runs, whereas Iter and Ant have much more difficulties,

No.	IterDSU	IterDHU	IterRand	AntDSU	AntDHU	AntRand	LSRand	LSDED
`60-*`	0.0 (0.0)	0.0 (0.0)	5.4 (1.0)	0.0 (0.0)	0.0 (0.0)	5.2 (0.9)	0.0 (0.1)	0.0 (0.0)
`65-*`	0.0 (0.0)	0.0 (0.0)	3.0 (0.7)	0.0 (0.0)	0.0 (0.0)	3.0 (0.7)	0.1 (0.1)	0.0 (0.0)
`70-*`	0.0 (0.0)	0.0 (0.0)	1.4 (0.5)	0.0 (0.0)	0.0 (0.0)	1.4 (0.6)	0.0 (0.1)	0.0 (0.0)
`75-*`	0.0 (0.0)	0.0 (0.0)	0.6 (0.4)	0.0 (0.0)	0.0 (0.0)	0.6 (0.4)	0.1 (0.2)	0.0 (0.0)
`80-*`	0.0 (0.0)	0.0 (0.0)	2.0 (0.9)	0.0 (0.0)	0.0 (0.0)	2.0 (0.8)	0.2 (0.3)	0.0 (0.0)
`85-*`	0.0 (0.0)	0.0 (0.0)	8.3 (1.7)	0.0 (0.0)	0.0 (0.0)	8.2 (1.6)	0.4 (0.6)	0.0 (0.1)
`90-*`	0.0 (0.0)	0.0 (0.0)	28.1 (2.5)	0.0 (0.0)	0.0 (0.0)	28.1 (2.4)	2.5 (1.2)	0.6 (0.6)

Table 4. Results for Iter, Ant, and LS on the first test suite instances (average costs on 100 runs, standard deviations in brackets).

particularly for those with highest utilization rates[1]. The effect of DED on LS is rather small compared to the effect of DHU and DSU on Ant and Iter, because DED is applied only once, whereas the heuristic in Ant and Iter is used for each generated sequence. Nevertheless, DED is quite helpful because it performs very well on these instances, as already discussed in section 3.

Finally, note that the results for IterDSU and IterDHU are much better than those reported for DSU and DHU in table 1, because Iter produces a large set of solutions and selects the best one, while the greedy approach produces only one solution for each run. Furthermore, Iter diversifies the search by choosing cars w.r.t. probabilities, which is beneficial when lots of solutions are generated.

6.3 Results for Test Suite 2

Table 5 presents the results for the second set of instances, the more difficult ones. The number of cycles of Ant and Iter is limited to 1 000, and the number of moves of LS to 200 000. Comparing Iter and Ant variants reveals that pheromone has a positive impact since Ant variants yield a better solution quality. In particular for Rand, the worst greedy heuristic, the use of pheromone makes a big difference. Thus, the key success factor of ACO is the employed greedy heuristic, demonstrated by the clear superiority of AntDSU and AntDHU variants. In addition to the heuristics, pheromone helps to improve solution quality, too.

Regarding the different greedy heuristics, DSU and DHU are comparable. It is remarkable that IterDSU and IterDHU achieve very good results. Although they are slightly inferior to their Ant counterparts, the obtained solution quality is comparable to LS. Using DED for the initial solution in LS is better than using a pure random sequence, coinciding to the results for the easy instances. However, LS seems to be slightly inferior to AntDHU and AntDSU. We believe that LS suffers from local optima, because we did not use mechanisms to escape

[1] We also made runs for IterRand and AntRand with 1000 cycles. In this case, on `90-*`, the average cost decreases to 23.9 for IterRand, and to 12.9 for AntRand. This shows that pheromone actually allows ants to improve the solution process, even though they do not reach LSRand performances.

No.	IterDSU	IterDHU	IterRand	AntDSU	AntDHU	AntRand	LSRand	LSDED
10-93	5.5 (0.6)	6.6 (0.7)	35.4 (1.7)	4.7 (0.5)	4.7 (0.5)	21.9 (1.7)	5.9 (1.1)	5.5 (0.9)
16-81	0.8 (0.4)	2.2 (0.5)	23.1 (1.1)	0.2 (0.4)	0.9 (0.4)	13.0 (1.2)	2.0 (0.8)	1.6 (0.8)
19-71	2.8 (0.4)	3.1 (0.4)	29.9 (1.7)	2.8 (0.4)	2.7 (0.4)	18.1 (1.8)	2.2 (0.4)	2.1 (0.3)
21-90	2.6 (0.5)	2.0 (0.0)	27.1 (1.5)	2.3 (0.5)	2.0 (0.1)	13.4 (1.5)	2.2 (0.4)	2.2 (0.4)
26-82	1.0 (0.2)	0.3 (0.5)	19.3 (1.3)	0.0 (0.0)	0.0 (0.0)	8.9 (1.1)	0.4 (0.5)	0.3 (0.5)
36-92	3.4 (0.6)	2.9 (0.4)	25.9 (1.7)	2.1 (0.3)	2.0 (0.0)	15.7 (1.2)	3.0 (0.5)	2.9 (0.6)
41-66	0.0 (0.0)	0.0 (0.0)	13.3 (1.3)	0.0 (0.0)	0.0 (0.0)	5.2 (0.9)	0.0 (0.2)	0.0 (0.1)
4-72	0.2 (0.4)	1.0 (0.4)	24.1 (1.6)	0.0 (0.0)	0.0 (0.0)	16.3 (1.4)	0.8 (0.6)	0.8 (0.6)
6-76	6.0 (0.0)	6.0 (0.0)	13.1 (0.9)	6.0 (0.0)	6.0 (0.0)	7.6 (0.6)	6.0 (0.0)	6.0 (0.0)

Table 5. Results for Iter, Ant, and LS on the second test suite instances (average costs on 100 runs, standard deviations in brackets).

from them. In our previous study for car sequencing problems involving several different types of constraints [8], we observed that solution quality can be improved significantly when using threshold accepting to escape from local optima. Thus, we think the gap between the LS variant we consider here and the best Ant variants could be closed by using an acceptance criterion like threshold accepting. Further, we remark that we have used the local search implementation from [8], which contains some unnecessary functionality for problems involving other constraint types. Thus, a specific implementation for the special car sequencing problem we consider here, is expected to be more efficient.

Three algorithms – IterDHU, AntDHU, and LSDED – are selected for further experiments regarding the impact of different CPU time limits. We successively limit Iter and Ant to 100, 500, 1 000, 2 500 and 10 000 cycles, which correspond to 20 000, 100 000, 200 000, 500 000, 2 000 000 moves for LS, respectively. The evolution of solutions' quality is shown in figure 1 for four representative instances. For small limits, LS is inferior to IterDHU and AntDHU. However, LS and AntDHU achieve quite similar results for the highest limit. Interestingly, IterDHU is less effective for these high limits. Thus, the pure sampling of iterated greedy is inferior to LS and the pheromone-driven AntDHU, supposed the latter variants are given enough time.

During all the runs, we found some new best solutions for instances of the second test suite. Table 6 compares previous results with the outcome of our experiments for the highest CPU time limit. The instances being reported as satisfiable have been solved by AntDHU and LSDED. Further, we found a feasi-

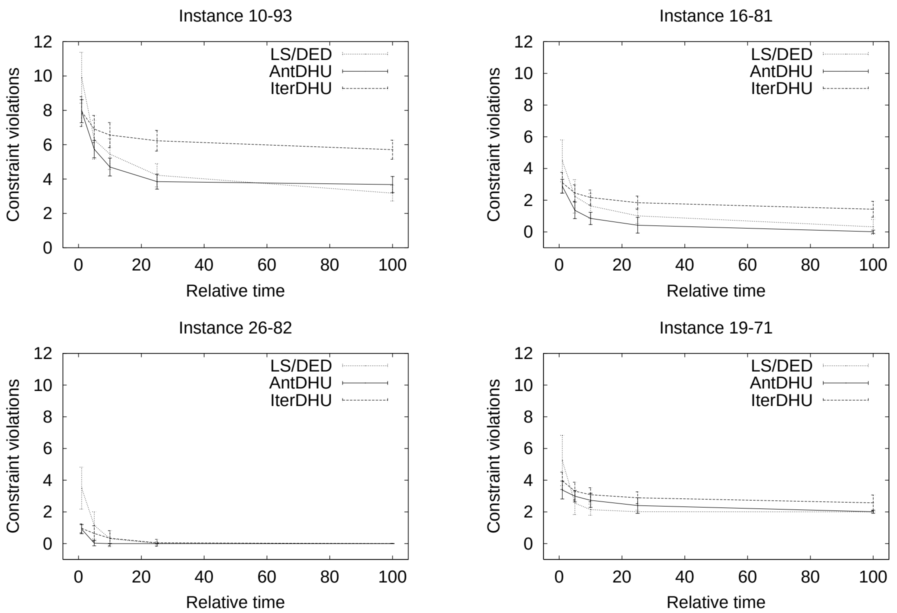

Fig. 1. Results on four representative benchmark instances of the second suite.

ble solution for the instance 26-82, for which it was formerly unknown whether it is satisfiable or not. The lower bound 2 on the number of constraint violations proved by Gent [5] for 19-71 is reached by all three algorithms.

Instance	10-93	16-81	19-71	21-90	26-82	36-92	41-66	4-72	6-76
Satisfiable?	no [9]	yes [6]	no [5, 9]	?	?	?	yes [6]	yes [9]	no [9]
Best solution	3	0	2	2	0	2	0	0	6
SR for IterDHU	0.00	0.00	0.43	1.00	1.00	0.70	1.00	0.65	1.00
SR for AntDHU	0.32	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00
SR for LSDED	0.85	0.68	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 6. Comparison with results reported in [5, 6, 9]: The best solution is the best value we found in all our experiments, and the success rate (SR) measures for each algorithm the ratio of runs that actually found the best value.

7 Conclusion

We evaluated several algorithms on two test suites of car sequencing problems. The first suite contains relatively easy instances since most of them could be solved even by greedy algorithms. The second suite contains some harder instances. Our best algorithms found solutions for the feasible ones among them, reached a lower bound for one infeasible instance, and proved one instance being feasible, for which it was formerly unknown whether it is satisfiable or not.

Among the greedy algorithms, the dynamic variants clearly outperform their static counterparts. Iterated greedy is surprisingly good, if good heuristics like DHU or DSU are employed. These algorithms solved all instances of the first suite in every run. But, pure sampling is inferior to more intelligent approaches like pheromone-driven ACO and LS, when more difficult instances of the second suite are considered and more CPU time is invested.

The best results are obtained by an ACO approach combined with a dynamic heuristic. The use of good heuristics is crucial and allows solving the first test suite in every run without using pheromone at all. On the second test suite, the results are improved when guiding search by pheromone. LS performs quite well, but it is slightly inferior to the best ACO variant for small CPU time limits; for larger limits, LS and the ACO approaches yield comparable solution quality. Although we employed a large neighbourhood, we still believe LS suffers from getting stuck in local optima. Therefore, we are confident that even better results can be obtained by using acceptance criteria like threshold accepting, as reported in [8] for problems involving different types of constraints.

Although the new results are significantly better than those obtained by a previous ACO algorithm [11], there are other algorithms from literature that deserve a comparison with our algorithms. In particular, we should compare our results with other local search approaches, which are using the Swap neighbourhood, repair heuristics and adaptive mechanisms to escape from local optima [1, 2, 7], and with genetic local search proposed in [14]. The difficulty is, however, that different machines, evaluation limits, and benchmarks have been used.

Some other issues are also open and should be the subject of further research. More specific, we want to check the effects of threshold accepting or restarting mechanisms on LS on the benchmarks used in this study. Further, it may be beneficial to integrate local search into the best ACO variant, by applying it to solutions constructed by ants. Since ACO turned out to be very effective, it is challenging to extend it such that it can cope with the real-world problems considered in [8].

References

1. A. Davenport and E. P. K. Tsang. Solving constraint satisfaction sequencing problems by iterative repair. In *Proceedings of the First International Conference on the Practical Applications of Constraint Technologies and Logic Programming*, 345 – 357, 1999
2. A. Davenport, E. Tsang, K. Zhu and C. Wang. GENET: a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI'94*, 325 – 330, 1994
3. M. Dincbas, H. Simonis and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of ECAI-88*, 290 – 295, 1988
4. M. Dorigo and G. Di Caro. The Ant Colony Optimization Meta-Heuristic. In D. Corne, M. Dorigo, and F. Glover (eds.), *New Ideas in Optimization*, 11 – 32, McGraw Hill, UK, 1999
5. I.P. Gent. Two Results on Car-sequencing Problems. Technical report APES. 1998
6. I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report APES-09-1999, available from `http://4c.ucc.ie/~tw/csplib`, a shorter version appeared in CP99, 1999
7. J.H.M. Lee, H.F. Leung and H.W. Won. Performance of a Comprehensive and Efficient Constraint Library using Local Search. In *Proceedings of 11th Australian Joint Conference on Artificial Intelligence*, 191 – 202, LNAI 1502, Springer, 1998
8. M. Puchta and J. Gottlieb. Solving Car Sequencing Problems by Local Optimization. In *Applications of Evolutionary Computing*, 132 – 142, LNCS 2279, Springer, 2002
9. J.-C. Regin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In *Principles and Practice of Constraint Programming*, 32 - 46, LNCS 1330, Springer, 1997
10. B. Smith. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In *Third Conference on the Practical Applications of Constraint Technology*, 321 – 330, 1996
11. C. Solnon. Solving Permutation Constraint Satisfaction Problems with Artificial Ants. In *Proceedings of ECAI-2000*, 118 – 122, IOS Press, 2000
12. T. Stützle and H.H. Hoos. MAX-MIN Ant System. *Journal of Future Generation Computer Systems*, Volume 16, 889 – 914, 2000
13. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993
14. T. Warwick and E. Tsang. Tackling car sequencing problems using a genetic algorithm. *Evolutionary Computation*, Volume 3, Number 3, 267 – 298, 1995