Journal of Mathematical Modelling and Algorithms **00:** 1–24, 2004.

An Ant Colony Optimization Algorithm for Shop Scheduling Problems ⋆

CHRISTIAN BLUM and MICHAEL SAMPELS
IRIDIA, Université Libre de Bruxelles, CP 194/6, Av. Franklin D. Roosevelt 50, 1050 Bruxelles, Belgium

Abstract. We deal with the application of ant colony optimization to group shop scheduling, which is a general shop scheduling problem that includes, among others, the open shop scheduling problem and the job shop scheduling problem as special cases. The contributions of this paper are twofold. First, we propose a neighborhood structure for this problem by extending the well-known neighborhood structure derived by Smutnicki and Nowicki for the job shop scheduling problem. Then, we develop an ant colony optimization approach, which uses a strong non-delay guidance for constructing solutions and which employs black-box local search procedures to improve the constructed solutions. We compare this algorithm to an adaptation of the tabu search by Nowicki and Smutnicki to group shop scheduling. Despite its general nature, our algorithm works particularly well when applied to open shop scheduling instances, where it improves the best known solutions for 15 of the 28 tested instances. Moreover, our algorithm is the first competitive ant colony optimization approach for job shop scheduling instances.

Mathematics Subject Classifications (2000):

Key words: Scheduling, group shop, job shop, open shop, ant colony optimization.

1. Introduction

Academic shop scheduling problems such as job shop scheduling (JSS) and open shop scheduling (OSS) are simplified models of scheduling problems often occurring in industrial settings. In general, these shop scheduling problems are $\mathcal{NP}$-hard and difficult to solve in practice, which justifies the need for efficient methods to obtain approximate solutions of high quality in a reasonable amount of time. Metaheuristics are successful algorithmic concepts to generate approximate solutions to $\mathcal{NP}$-hard optimization problems. During the last decade many researchers have successfully tried to apply metaheuristics to shop scheduling problems. However, due to the quite different characteristics of the different shop scheduling problems, existing approaches are often too specialized and generally cannot be adapted such

⋆ This work was supported by the "Metaheuristics Network", a Research Training Network funded by the Improving Human Potential program of the CEC, grant HPRN-CT-1999-00106. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

that they work well when applied to other shop scheduling problems. Therefore, in this work we propose a metaheuristic approach, namely an ant colony optimization (ACO) approach, to tackle a more general shop scheduling problem called the *group shop scheduling* (*GSS*). The GSS problem includes, among others, the well-known open shop scheduling (OSS) problem and the job shop scheduling (JSS) problem as special cases.

Solution Techniques for Shop Scheduling Problems. Especially for the JSS problem there are many excellent solution techniques to be found in the literature. Metaheuristics such as tabu search (TS) approaches [14, 32] and local search based approaches, such as the one proposed in [3], based on the *shifting bottleneck* procedure [1] have been very successful. These approaches excel others not necessarily in solution quality, but almost always in computation time. Other metaheuristic approaches that can not compete with the above-mentioned approaches in terms of efficiency, but that work well when computation time is of no concern, are an evolutionary computation (EC) approach [25] and a simulated annealing (SA) approach [40]. More recently, constraint propagation methods have proven to be very successful [8, 20]. For an overview of different JSS solution techniques, see [4, 26].

The research activities aimed at tackling the OSS problem have long been dominated by exact methods such as branch & bound [9, 19]. In fact, the algorithm proposed in [19] is a state-of-the-art algorithm for small and medium size problem instances. Early metaheuristic approaches such as the EC approach outlined in [21] were not very successful. The first quite successful algorithm was a TS approach proposed in [28], which was improved later by an EC approach proposed in [29]. Recently, a new state-of-the-art algorithm for OSS has been accepted for publication [6]. This algorithm is a hybrid between ant colony optimization and beam search.

Ant Colony Optimization Applied to Scheduling Problems. Ant colony optimization (ACO) [16] is a metaheuristic approach to tackle hard combinatorial optimization problems. The main idea of ACO is to use a parametrized probabilistic model to construct solutions that are then used to update the model parameter values with the aim of increasing the probability of constructing high quality solutions. In every iteration, a number of agents (artificial ants) construct solutions by probabilistically making a number of local decisions. ACO has been proven a successful technique for numerous $\mathcal{NP}$-hard combinatorial optimization problems. In the field of scheduling, ACO has been successfully applied to the single machine weighted tardiness (SMWT) problem [15], the flow shop scheduling (FSS) problem [36], and the resource constraint project scheduling (RCPS) problem [30]. However, the application to shop scheduling problems – in particular JSS and OSS – has proven quite difficult.

Related Work. The first ACO algorithm to tackle a shop scheduling problem was the one by Colorni et al. [12] to tackle the JSS problem. The performance of this algorithm was far from reaching state-of-the-art performance. A first attempt to develop an ACO algorithm to tackle the OSS problem was made in [33]. However, the experimental evaluation was quite limited. In [5] we proposed a first successful algorithm to tackle the GSS problem. The results – especially when applied to OSS instances – were still quite far from the performance of state-of-the-art algorithms.

Dorndorf and Pesch [18] proposed a genetic algorithm that is based on learning the priority rules to be used at each step of constructing a solution. One of the main differences between this algorithm and our ACO approach lies in the fact that our algorithm learns – assisted by the priority rules – which operation to choose for each construction step. An advantage of our algorithm is that the construction of an optimal solution is never excluded, which might be the case in the algorithm by Dorndorf and Pesch. Furthermore, our algorithm obtains better computational results.

The outline of the paper is as follows. In Section 2, we outline the group shop scheduling problem. In Section 3, we introduce a neighborhood structure for this problem before we outline our ant colony optimization approach to the group shop scheduling problem in Section 4. Section 5 is dedicated to the experimental evaluation of the ACO approach. First, we propose a new set of benchmark instances. Then, we fine-tune the solution construction mechanism of the ACO algorithm before we present experimental results. We compare the ACO approach to an adaptation of the well-known TS approach by Nowicki and Smutnicki [32] to group shop scheduling. Finally, in Section 6, we provide a summary and conclusions.

2. Group Shop Scheduling

In shop scheduling problems, *jobs* (respectively, *operations*) are to be processed by *machines* with the objective of minimizing some function of the completion times of the jobs. In the following we give a formal description of a general shop scheduling problem, called the *group shop scheduling* (*GSS*) *problem*, that includes the JSS problem as well as the OSS problem. This problem was introduced – under the name FOP Shop Scheduling – in 1997 by the TU Eindhoven [39] as the subject of a mathematics contest.

The GSS problem may be formulated as follows. Given is a set of operations $\mathcal{O} = \{o_1, \ldots, o_n\}$. Each operation $o \in \mathcal{O}$ has a processing time $p(o)$. The structure of the problem is defined as follows:

- Set $\mathcal{O}$ is partitioned into subsets $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_{|\mathcal{M}|}\}$. The operations in $\mathcal{M}_i \in \mathcal{M}$ have to be processed by the same machine. For the sake of simplicity we identify each set $\mathcal{M}_i \in \mathcal{M}$ of operations with the machine they have to be processed by, and call $\mathcal{M}_i$ a machine.
- Set $\mathcal{O}$ is also partitioned into subsets $\mathcal{J} = \{\mathcal{J}_1, \ldots, \mathcal{J}_{|\mathcal{J}|}\}$, where each set of operations $\mathcal{J}_j \in \mathcal{J}$ is called a job.
- Furthermore, given is a refinement of the job-partition $\mathcal{J}$ into a group-partition $\mathcal{G} = \{\mathcal{G}_1, \ldots, \mathcal{G}_{|\mathcal{G}|}\}$, where each set of operations $\mathcal{G}_l \in \mathcal{G}$ is called a group.

Note that all operations belonging to a certain group belong to the same job. This holds true because the group-partition is a *refinement* of the job-partition.

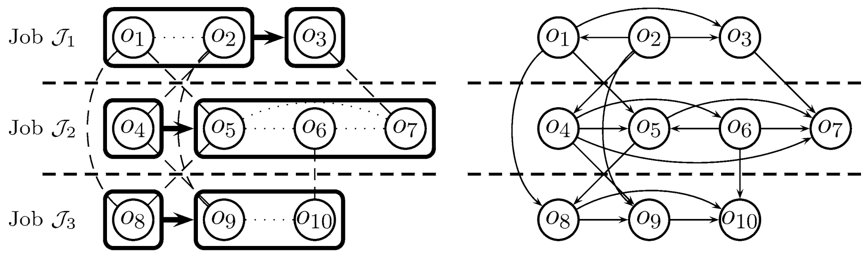
– The groups of each job are linearly ordered, i.e., given is a permutation of the groups of each job. If, in such a permutation, a group $\mathcal{G}_l$ comes before a group $\mathcal{G}_k$, we write $\mathcal{G}_l \preceq \mathcal{G}_k$. Equally, we write for all $o \in \mathcal{G}_l$ and all $o' \in \mathcal{G}_k$ that $o \preceq o'$, which means that the processing of operation o has to be finished before the processing of operation o' can be started. The set of *predecessors* of an operation $o \in \mathcal{O}$ is given by $\text{pred}(o) \leftarrow \{o' \in \mathcal{O} \mid o' \preceq o\}$. Furthermore, given two operations $o', o \in \mathcal{O}$ with $o' \preceq o$, o' is called a *direct* predecessor of o, denoted by $o' \preceq_d o$, if an operation $o'' \in \mathcal{O}$ such that $o' \preceq o'' \preceq o$ does not exist.

Therefore, each operation $o \in \mathcal{O}$ has to be processed by a machine $m(o) \in \mathcal{M}$, belongs to a job $j(o) \in \mathcal{J}$ and belongs to a group $g(o) \in \mathcal{G}$. In this paper we consider the case where each machine can process at most one operation at a time. Operations must be processed without preemption (that is, once the process of an operation has started it must be completed without interruption).

A solution to an instance of the GSS problem is given by permutations $\pi^{\mathcal{M}_i}$ of the operations in $\mathcal{M}_i, \forall i \in \{1, \ldots, |\mathcal{M}|\}$, and permutations $\pi^{\mathcal{G}_l}$ of the operations in $\mathcal{G}_l, \forall l \in \{1, \ldots, |\mathcal{G}|\}$. These permutations define processing orders for all machines $\mathcal{M}_i$ and all groups $\mathcal{G}_l$. Note that not all combinations of permutations are feasible, because some combinations of permutations might define cycles in the processing orders. There are several possibilities to measure the cost (i.e., to define the objective function) of a solution. In this paper we deal with an objective function known as *makespan*. The goal is to find a solution with minimum makespan. The makespan of a solution is the time it takes all the operations of an instance to be processed, assuming the processing of the first operation(s) starts at time zero. The formal definition of the makespan of a solution depends on the solution representation that is used. In order to refer to a solution which may be given in any format, we use the notifier s, with $C_{\max}(s)$ denoting the makespan of s.

2.1. DISJUNCTIVE GRAPHS

A very popular way to depict shop scheduling instances is the *disjunctive graph* [35] $G_{\text{dis}} = (V, A, E)$, where V is the set of nodes, A is the set of conjunctive (directed) arcs, and E is the set of disjunctive (undirected) arcs. Given an instance of the GSS problem, the disjunctive graph G_{dis} is obtained as follows: For each operation $o \in \mathcal{O}$, a node $v_o \in V$ is introduced. In the following we identify the nodes of G_{dis} with the corresponding operations. Furthermore, for each pair of operations $o, o' \in \mathcal{O}$ with $o \preceq_d o'$, a conjunctive arc $a_{o,o'} \in A$ is introduced. Finally, for each pair of operations $o, o' \in \mathcal{O}$ with either $m(o) = m(o')$ or $g(o) = g(o')$, a disjunctive arc $e_{o,o'} \in E$ is introduced. Figure 1(a) shows the disjunctive graph of a simple GSS instance with 10 operations partitioned into 3 jobs, 4 machines and 6 groups: $\mathcal{O} = \{o_1, \ldots, o_{10}\}$, $\mathcal{J} = \{\mathcal{J}_1 = \{o_1, o_2, o_3\}, \mathcal{J}_2 = \{o_4, \ldots, o_7\},$



(a) GSS problem instance. (b) A feasible solution.

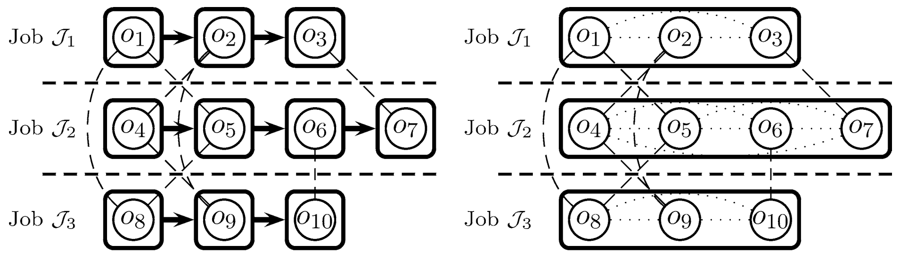
Figure 1. (a) The disjunctive graph of a simple GSS instance. The conjunctive arcs are simplified as inter-group arcs. Furthermore, the disjunctive arcs are shown as dashed (between pairs of operations from the same machine), respectively dotted (between pairs of operations from the same group), lines. (b) A feasible solution of the problem instance shown in (a). The disjunctive arcs are directed and the resulting directed graph does not contain any cycles, which means that there are no cycles in the processing orders.

$\mathcal{J}_3 = \{o_8, o_9, o_{10}\}\}$, $\mathcal{G} = \{\mathcal{G}_1 = \{o_1, o_2\}, \mathcal{G}_2 = \{o_3\}, \mathcal{G}_3 = \{o_4\}, \mathcal{G}_4 = \{o_5, o_6, o_7\}, \mathcal{G}_5 = \{o_8\}, \mathcal{G}_6 = \{o_9, o_{10}\}$, $\mathcal{M} = \{\mathcal{M}_1 = \{o_1, o_5, o_8\}, \mathcal{M}_2 = \{o_2, o_4, o_9\}, \mathcal{M}_3 = \{o_3, o_7\}, \mathcal{M}_4 = \{o_6, o_{10}\}\}$. One way of representing solutions to a GSS problem instance is the precedence graph. This graph is obtained by directing the disjunctive arcs of the disjunctive graph according to the machine-permutations and the group-permutations as given by a solution. Figure 1(b) shows a precedence graph that corresponds to a feasible solution to the GSS problem instance that is shown in Figure 1(a). The makespan of a solution (in the form of a precedence graph) is defined as the length of the longest directed path in the graph. Such a path is commonly called a *critical path*. The length of a critical path is given by the sum of the processing times of the operations on that path. Note that there might be more than one critical path in a solution. Let us assume the following processing times for the operations of the GSS problem instance that is shown in Figure 1(a).

Operation	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_{10}
Processing time	1	3	5	4	3	1	6	2	1	3

Then, the critical path of the solution that is shown in Figure 1(b) is o_2-o_4-o_6-o_5-o_7. Therefore, the makespan, i.e., the objective function value, of this solution is $3 + 4 + 1 + 3 + 6 = 17$.

As mentioned above, the GSS problem formulation contains the OSS problem and the JSS problem as extreme cases. Each GSS problem instance that is characterized by the fact that each operation is in its own group is also a JSS problem instance. Furthermore, each GSS problem instance that is characterized by the fact that the job-partition is equal to the group-partition is also an OSS problem instance. As an example, Figure 2(a) shows a JSS version of the GSS problem instance that is shown in Figure 1(a), whereas Figure 2(b) shows the OSS version.



(a) JSS problem instances. (b) OSS problem instance.

Figure 2. (a) A JSS version of the GSS problem instance that is shown in Figure 1(a). The instance is a JSS instance, because each operation is in its own group. (b) The OSS version of the GSS problem instance that is shown in Figure 1(a). The instance is an OSS instance because the job-partition is equal to the group-partition. In other words: all the operations of a job are in one group.

2.2. PERMUTATION REPRESENTATION

Some permutations of all operations correspond to feasible solutions of GSS problem instances. This is due to the fact that a permutation of all operations contains the machine-permutations and the group-permutations. As an example consider permutation $(o_2, o_1, o_4, o_6, o_5, o_8, o_9, o_{10}, o_3, o_7)$ of the 10 operations of the GSS problem instance that is shown in Figure 1(a). This permutation corresponds to the solution of this problem instance that is shown in Figure 1(b). This is because it induces permutation (o_2, o_1) for group $\mathcal{G}_1$, (o_6, o_5, o_7) for group $\mathcal{G}_4$, and (o_9, o_{10}) for group $\mathcal{G}_6$. Furthermore, it induces permutation (o_1, o_5, o_8) for machine $\mathcal{M}_1$, (o_2, o_4, o_9) for machine $\mathcal{M}_2$, (o_3, o_7) for machine $\mathcal{M}_3$, and (o_6, o_{10}) for machine $\mathcal{M}_4$. However, note that there is generally a many-to-one mapping from the set of feasible permutations of all operations to the set of different solutions of a GSS problem instance.

List scheduler algorithms [22] are often applied in constructing solutions of shop scheduling problems in permutation form. They are easy to implement and their complexity is rather low. In the following, we denote solutions consisting of a sequence of solution components by $\mathfrak{s}$. In GSS, each operation $o \in \mathcal{O}$ is regarded as a solution component. Furthermore, partial solutions (in terms of partial permutations) are denoted by $\mathfrak{s}^p$.

A list scheduler algorithm builds a permutation of all operations from left to right by appending at each of $n = |\mathcal{O}|$ construction steps an operation from a set $\mathcal{O}_t$ of allowed operations to the current partial permutation. Set $\mathcal{O}_t$ is defined as follows. At each step $t \in \{1, \ldots, n\}$ the set $\mathcal{O}$ of operations is partitioned into set $\mathcal{O}^-$, which are the operations that are already in the partial solution, and set $\mathcal{O}^+$, which are the operations that still have to be dealt with. In order to exclusively

ALGORITHM 1. List scheduler algorithm for the GSS problem

$\mathfrak{s}^p = \langle\rangle$
$\mathcal{O}^+ \leftarrow \mathcal{O}$
for $t = 1, \ldots, |\mathcal{O}|$
  $\mathcal{O}_t \leftarrow \{o \in \mathcal{O}^+ \mid \text{pred}(o) \cap \mathcal{O}^+ = \emptyset\}$
  $\mathcal{O}_t' \leftarrow \mathsf{Restrict}(\mathfrak{s}^p, \mathcal{O}_t)$
  $o^* \leftarrow \mathsf{Choose}(\mathcal{O}_t{}')$
  $\mathfrak{s}^p \leftarrow$ extend $\mathfrak{s}^p$ byappending operation o^*
  $\mathcal{O}^+ \leftarrow \mathcal{O}^+ \setminus \{o^*\}$
end for

generate feasible solutions, $\mathcal{O}_t$ is defined – in construction step t – as a subset of $\mathcal{O}^+$ in the following way:

$$\mathcal{O}_t \leftarrow \{o \in \mathcal{O}^+ \mid \text{pred}(o) \cap \mathcal{O}^+ = \emptyset\}, \tag{1}$$

which means that in each construction step an operation can only be chosen if all its predecessors are already in the partial solution. The algorithmic framework of a list scheduler algorithm is shown in Algorithm 1. In each construction step, candidate list strategies, implemented in function $\mathsf{Restrict}(\mathfrak{s}^p, \mathcal{O}_t)$, may be applied to further restrict set $\mathcal{O}_t$. In the following we outline two strategies proposed by Giffler and Thompson [22]. The first one works as shown in Algorithm 2. The partition of the set of operations $\mathcal{O}$, with respect to a partial solution $\mathfrak{s}^p$ into $\mathcal{O}^+$ and $\mathcal{O}^-$, induces a partition of the operations of every job $\mathcal{J}_j \in \mathcal{J}$ into $\mathcal{J}_j{}^+$ and $\mathcal{J}_j{}^-$ and of every machine $\mathcal{M}_i \in \mathcal{M}$ into $\mathcal{M}_i{}^+$ and $\mathcal{M}_i{}^-$. First, the earliest possible completion times $t_{ec}(o, \mathfrak{s}^p)$ of all the operations in $\mathcal{O}_t$ are calculated. This can be done by deriving the partial schedule⋆ that is defined by the partial solution $\mathfrak{s}^p$. Then, one of the machines $\mathcal{M}^*$, with minimal completion time t^*, is chosen and set $\mathcal{O}_t{}'$ is defined as the set of all operations of $\mathcal{O}_t$ which need to be processed by machine $\mathcal{M}^*$ and whose earliest possible starting time is before t^*. This way of restricting set $\mathcal{O}_t$ produces active schedules.⋆⋆ Algorithm 1, using the candidate list strategy as given by Algorithm 2, is commonly called *GT algorithm*.

Another way of implementing $\mathsf{Restrict}(\mathfrak{s}^p, \mathcal{O}_t)$ is presented in Algorithm 3. It works as follows: First, the earliest possible starting time t^*, among all operations in $\mathcal{O}_t$, is determined. Then, $\mathcal{O}_t$ is restricted to all operations that can start at time t^*. By this way of restricting set $\mathcal{O}_t$, permutations are generated that correspond to

⋆ Note that a *schedule* is yet another way of representing solutions of shop scheduling problems in which each operation is given a starting time. Schedules can be derived from (partial) permutations of operations in a straight-forward way.

⋆⋆ A feasible schedule is called active if it is not possible to construct another schedule by changing the processing orders on the machines or in the groups and having at least one operation starting earlier and no operation finishing later.

ALGORITHM 2. Restrict($\mathfrak{s}^p$, $\mathcal{O}_t$) method for producing active schedules

input: $\mathfrak{s}^p$, $\mathcal{O}_t$

Determine $t^* \leftarrow \min\{t_{ec}(o, \mathfrak{s}^p) \mid o \in \mathcal{O}_t\}$

$\mathcal{M}^* \leftarrow$ Select randomly from $\{\mathcal{M}_i \in \mathcal{M} \mid {\mathcal{M}_i}^+ \cap \mathcal{O}_t \neq \emptyset,\ \exists o \in {\mathcal{M}_i}^+$ with $t_{ec}(o, \mathfrak{s}^p) = t^*\}$

${\mathcal{O}_t}' \leftarrow \{o \in \mathcal{O}_t \mid o \in \mathcal{M}^*$ and $t_{es}(o, \mathfrak{s}^p) < t^*\}$

output: restricted set ${\mathcal{O}_t}'$

ALGORITHM 3. Restrict($\mathfrak{s}^p$, $\mathcal{O}_t$) method for producing non-delay schedules

input: $\mathfrak{s}^p$, $\mathcal{O}_t$

Determine $t^* \leftarrow \min\{t_{es}(o, \mathfrak{s}^p) \mid o \in \mathcal{O}_t\}$

${\mathcal{O}_t}' \leftarrow \{o \in \mathcal{O}_t \mid t_{es}(o, \mathfrak{s}^p) = t^*\}$

output: restricted set ${\mathcal{O}_t}'$

non-delay schedules.⋆ Algorithm 1 using the candidate list strategy as given by Algorithm 3 is commonly called *ND algorithm*.

The function Choose(${\mathcal{O}_t}'$) for selecting an operation from ${\mathcal{O}_t}'$ is often implemented by means of *priority rules* or *dispatching rules*. Table I shows a selection thereof. Usually the priority rules are used in a deterministic manner. However, they may also be used probabilistically (e.g., in a roulette-wheel-selection manner) instead of deterministically. None of these rules can be labelled the "best-performing" priority rule. Which rule performs best strongly depends on the structure of the problem instance to be solved. An extensive overview of priority rules is provided in [24].

Table I. Different priority (or dispatching) rules

Rule	Description
Random	an operation chosen at random
EST	an operation having the earliest starting time
EFT	an operation having the earliest finishing time
SPT	an operation having the shortest processing time
LPT	an operation having the longest processing time
LWR	an operation having the least work remaining in the job
MWR	an operation having the most work remaining in the job
LTW	an operation having the least total work in the job
MTW	an operation having the most total work in the job

⋆ A feasible schedule is called non-delay if no machine is kept idle while an operation is waiting for processing. Non-delay schedules are a subset of active schedules.

3. A Neighborhood Structure for GSS

As outlined in Section 2.1, a directed path $\mathfrak{P}$ in the precedence graph that corresponds to a feasible solution s is called a *critical path*, if and only if $\sum_{o \in \mathfrak{P}} p(o) = C_{\max}(s)$. $\mathcal{M}$ induces a subdivision on a critical path $\mathfrak{P} = (o_1, \ldots, o_q)$ into *machine blocks* of consecutive operations belonging to the same machine, as $\mathcal{G}$ induces a subdivision into *group blocks* of consecutive operations belonging to the same group. Brucker et al. [10] proved for the JSS problem that if there is a feasible solution s' with $C_{\max}(s') < C_{\max}(s)$, then there is a machine block $B^i_M = (o^i_1, \ldots, o^i_{m_i})$ on a critical path $\mathfrak{P}$ of s such that either $\exists o \in B^i_M$, $o \neq o^i_1$ with $o \preceq^{s'} o^i_1$, or $\exists o \in B^i_M$, $o \neq o^i_{m_i}$ with $o^i_{m_i} \preceq^{s'} o$, where $o \preceq^{s'} o'$ means that o has to be processed before o' in s' (with $o, o' \in \mathcal{O}$). For the GSS problem, we generalize the above result.

THEOREM 1. *Let s be a feasible solution of a GSS instance. If there exists a feasible solution s' with $C_{\max}(s') < C_{\max}(s)$, then there exists a machine or a group block $B^i = (o^i_1, \ldots, o^i_{n_i})$ on a critical path $\mathfrak{P}$ in s, such that either $\exists o \in B^i$, $o \neq o^i_1$ with $o \preceq^{s'} o^i_1$, or $\exists o \in B$, $o \neq o^i_{n_i} \in B$ with $o_{n_i} \preceq^{s'} o$.*

Proof. Let $\mathfrak{P}$ be a critical path in s. Let $B^i_M = (o^i_1, \ldots, o^i_{m_i})$ denote the i-th machine block and $B^j_G = (\bar{o}^j_1, \ldots, \bar{o}^j_{g_j})$ be the j-th group block on $\mathfrak{P}$. Let k_M, respectively k_G, denote the total number of machine blocks, respectively group blocks. Assume that there is a feasible solution s' with $C_{\max}(s') < C_{\max}(s)$ and no operation of any group or machine block in $\mathfrak{P}$ is processed in s' before the first or after the last operation of the corresponding block. Then $\forall i \in \{1, \ldots, k_M\}$ it holds that

$$o^i_1 \preceq^{s'} o^i_l \quad \forall l \in \{1, \ldots, m_i\} \quad \text{and} \quad o^i_l \preceq^{s'} o^i_{m_i} \quad \forall l \in \{1, \ldots, m_i\}, \tag{2}$$

and $\forall j \in \{1, \ldots, k_G\}$ it holds that

$$\bar{o}^j_1 \preceq^{s'} \bar{o}^j_l \quad \forall l \in \{1, \ldots, g_j\} \quad \text{and} \quad \bar{o}^j_i \preceq^{s'} \bar{o}^j_{g_j} \quad \forall l \in \{1, \ldots, g_j\}. \tag{3}$$

Therefore, the precedence graph that corresponds to solution s' contains a path

$$(o^1_1, u^1_2, \ldots, u^1_{m_1-1}, o^1_{m_1}, \ldots, o^{k_M}_1, u^{k_M}_2, \ldots, u^{k_M}_{m_{k_M}-1}, o^{k_M}_{m_{k_M}}), \tag{4}$$

where $(u^i_2, \ldots, u^i_{m_i-1})$ is a permutation of $(o^i_2, \ldots, o^i_{m_i-1})$. Furthermore, the precedence graph that corresponds to solution s' contains a path

$$(\bar{o}^1_1, \bar{u}^1_2, \ldots, \bar{u}^1_{g_1-1}, \bar{o}^1_{g_1}, \ldots, \bar{o}^{k_G}_1, \bar{u}^{k_G}_2, \ldots, \bar{u}^{k_G}_{g_{k_G}-1}, \bar{o}^{k_G}_{g_{k_G}}), \tag{5}$$

where $(\bar{u}^j_2, \ldots, \bar{u}^j_{g_j-1})$ is a permutation of $(\bar{o}^j_2, \ldots, \bar{o}^j_{g_j-1})$. By identifying $u^i_1 = o^i_1$, $u^i_{m_i} = o^i_{m_i}$, $\bar{u}^j_1 = \bar{o}^j_1$, $\bar{u}^j_{g_j} = \bar{o}^j_{g_j}$, we get

$$C_{\max}(s') \geqslant \sum_{i=1}^{k_M} \sum_{l=1}^{m_i} p(o^i_l) = \sum_{i=1}^{k_M} \sum_{l=1}^{m_i} p(u^i_l) = C_{\max}(s) \tag{6}$$

and

$$C_{\max}(s') \geqslant \sum_{j=1}^{k_G} \sum_{l=1}^{g_j} p(\bar{o}_l^j) = \sum_{j=1}^{k_G} \sum_{l=1}^{g_j} p(\bar{u}_l^j) = C_{\max}(s) \tag{7}$$

which is a contradiction of the assumption. □

Given this theorem, it is reasonable to define a neighborhood of a feasible solution s of a GSS instance as an extension of the neighborhood structure proposed by Nowicki and Smutnicki for the JSS problem [32]. The new neighborhood, henceforth denoted by $\mathcal{N}_{1c,\text{GSS}}$, is defined as follows: A feasible solution s' is a neighbor of a feasible solution s (i.e., $s' \in \mathcal{N}_{1c,\text{GSS}}(s)$) if, in a critical path $\mathfrak{P}$ of s for exactly one machine block or exactly one group block $B = (o_1, o_2, \ldots, o_{n_k-1}, o_{n_k})$ on $\mathfrak{P}$, the order of o_1 and o_2 or the order of o_{n_k-1} and o_{n_k} is swapped in s'. Hence the first two operations of the first block in $\mathfrak{P}$ and the last two operations in the last block of $\mathfrak{P}$ are excluded.

4. Ant Colony Optimization for GSS

Our ACO approach, henceforth denoted by ACO_GSS, is a $\mathcal{MAX}$-$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) in the hyper-cube framework (HCF), as proposed by Blum and Dorigo [7]. $\mathcal{MM}$AS is an improvement of the original ant system (AS), which was proposed by Dorigo et al. [17]. $\mathcal{MM}$AS differs from AS by applying a lower and an upper bound, $\tau_{\min}$ and $\tau_{\max}$, to the pheromone values. The lower bound (a small positive constant) prevents the algorithm from converging* toward a solution. The HCF [7] is characterized by a pheromone update rule that limits the pheromone values to the interval [0, 1]. This has some theoretical as well as practical implications. For example, an ACO algorithm that is implemented in the HCF is likely to be more robust than a standard ACO algorithm.

One of the most important components of an ACO algorithm is the pheromone model. For ACO_GSS we use a pheromone model, henceforth referred to as *relation-learning model*. In this model, two operations o_i and o_j are called *related* if they are either in the same group or if they have to be processed by the same machine. We denote the set of operations that are related to an operation o_i by $\mathcal{R}_i$. Then, the relation-learning model consists of a pheromone trail parameter $\mathcal{T}_{ij}$ and a pheromone trail parameter $\mathcal{T}_{ji}$ for each pair of *related* operations $o_i, o_j \in \mathcal{O}$. The value τ_{ij} of pheromone trail parameter $\mathcal{T}_{ij}$ encodes the desirability of processing o_i before o_j, whereas the value τ_{ji} of pheromone trail parameter $\mathcal{T}_{ji}$ encodes the desirability of processing o_j before o_i. As before, the set of all pheromone trail parameters is denoted by $\mathcal{T}$.

* In the course of this work we refer to convergence in the sense of stochastic convergence, see also [37].

In the following we outline ACO_GSS. A high level description of this algorithm is given in Algorithm 4. The data structures used by this algorithm, in addition to counters and the already defined pheromone trails $\mathcal{T}$, are:

- the *iteration-best* solution $\mathfrak{s}_{ib}$: the best solution generated in the current iteration by the n_a ants;
- the *best-so-far* solution $\mathfrak{s}_{bs}$: the best solution generated since the start of the algorithm;
- the *restart-best* solution $\mathfrak{s}_{rb}$: the best solution generated since the last restart of the algorithm;
- the *convergence factor cf*, $0 \leqslant cf \leqslant 1$: a measure of how far the algorithm is from convergence;
- the Boolean variable *bs_update*: it becomes true when the algorithm reaches convergence.

A high level description of the algorithm is as follows (note that the main procedures are explained in detail later on). First, all the variables are initialized. In particular, the pheromone values are set to the initial value 0.5 by the procedure InitializePheromoneValues($\mathcal{T}$). Second, the n_a ants apply the ConstructSolution($\mathcal{T}$) procedure to construct n_a solutions. These solutions are then improved by the application of the ApplyLocalSearch($\mathfrak{S}_{iter}$) procedure. Third, the value of the variables $\mathfrak{s}_{ib}$, $\mathfrak{s}_{rb}$ and $\mathfrak{s}_{bs}$ is updated (note that, until the first restart of the algorithm, $\mathfrak{s}_{rb}$ represents the same solution as $\mathfrak{s}_{bs}$). Fourth, pheromone trail values are updated via the ApplyPheromoneUpdate(cf, bs_update, $\mathcal{T}$, $\mathfrak{s}_{rb}$, $\mathfrak{s}_{bs}$) procedure. Fifth, a new value for the convergence factor *cf* is computed. Depending on this value, as well as on the value of the Boolean variable *bs_update*, a decision on whether to restart the algorithm or not is made. For the case where the algorithm is restarted, the procedure ResetPheromoneValues($\mathcal{T}$) is applied and all the pheromones are reset to their initial value (0.5). The algorithm is iterated until some opportunely defined termination conditions are satisfied. Once terminated the algorithm returns the best-so-far solution $\mathfrak{s}_{bs}$. The components of that algorithm are outlined in more detail below.

DetermineNumberOfAnts(P): After preliminary tests, the number of ants used per iteration was set depending on the problem instance under consideration:

$$n_a \leftarrow \max\left\{10, \left\lfloor \frac{|\mathcal{O}|}{10} \right\rfloor\right\}. \tag{8}$$

ConstructSolution($\mathcal{T}$): Each ant constructs a solution by using the list scheduler algorithm that is explained in Algorithm 1. In order to fully define this algorithm, it has to be specified how the functions Restrict($\mathfrak{s}^p$, $\mathcal{O}_t$) and Choose($\mathcal{O}_t'$) are implemented.

The function Restrict($\mathfrak{s}^p$, $\mathcal{O}_t$) may restrict set $\mathcal{O}_t$, which contains the allowed operations for extending the current partial solution $\mathfrak{s}^p$, by means of candidate list strategies. We considered three possibilities for implementing this function: (i) no

ALGORITHM 4. ACO for the GSS problem (ACO_GSS)

input: A problem instance P of the GSS problem
$\mathfrak{s}_{bs} \leftarrow$ NULL, $\mathfrak{s}_{rb} \leftarrow$ NULL, $cf \leftarrow 0$, $bs\_update \leftarrow$ FALSE
$n_a \leftarrow$ DetermineNumberOfAnts(P)
InitializePheromoneValues$(\mathcal{T})$
while termination conditions not satisfied **do**
  $\mathfrak{S}_{iter} \leftarrow \emptyset$
  for $j = 1$ to n_a **do**
    $\mathfrak{S}_{iter} \leftarrow \mathfrak{S}_{iter} \cup$ ConstructSolution$(\mathcal{T})$
  end for
  ApplyLocalSearch$(\mathfrak{S}_{iter})$
  $\mathfrak{s}_{ib} \leftarrow$ argmin$\{C_{\max}(\mathfrak{s}) \mid \mathfrak{s} \in \mathfrak{S}_{iter}\}$
  EliteAction$(\mathfrak{s}_{ib})$
  Update$(\mathfrak{s}_{ib}, \mathfrak{s}_{rb}, \mathfrak{s}_{bs})$
  ApplyPheromoneUpdate$(cf, bs_update, \mathcal{T}, \mathfrak{s}_{rb}, \mathfrak{s}_{bs})$
  $cf \leftarrow$ ComputeConvergenceFactor$(\mathcal{T})$
  if $cf > 0.99$ **then**
    if $bs\_update =$ TRUE **then**
      ResetPheromoneValues$(\mathcal{T})$
      $\mathfrak{s}_{rb} \leftarrow$ NULL
      $bs\_update \leftarrow$ FALSE
    else
      $bs\_update \leftarrow$ TRUE
    end if
  end if
end while
output: $\mathfrak{s}_{bs}$

restriction of set $\mathcal{O}_t$ at all (henceforth denoted by NR construction), (ii) restriction of $\mathcal{O}_t$ by means of the GT mechanism as given in Algorithm 2 (henceforth denoted by GT construction), and (iii) restriction of $\mathcal{O}_t$ by means of the ND mechanism as given in Algorithm 3 (henceforth denoted by ND construction). In Section 5.2 we experimentally determine the best choice.

Furthermore, in function Choose$(\mathcal{O}_t')$ an operation from set $\mathcal{O}_t{}'$ is chosen for extending the current partial solution $\mathfrak{s}^p$. This is done according to the following transition probabilities:

$$\mathbf{p}(o_i \mid \mathcal{T}) = \frac{\min_{o_j \in \mathcal{R}_i \cap \mathcal{O}^+} \tau_{ij} \cdot \eta(o_i)^\beta}{\sum_{o_k \in \mathcal{O}_t{}'} \min_{o_j \in \mathcal{R}_k \cap \mathcal{O}^+} \tau_{kj} \cdot \eta(o_k)^\beta}, \quad \forall o_i \in \mathcal{O}_t{}', \tag{9}$$

where $\mathcal{O}^+$ is the set of operations that are not scheduled yet, and $\eta(o_i)$ denotes the heuristic information for an operation o_i, whose weight is adjusted by the parameter β. After preliminary tests we used a setting of $\beta = 10$ for all our experiments. In Section 5.2 we test eight different settings for the heuristic information.

ApplyLocalSearch($\mathfrak{S}_{iter}$): To every solution $\mathfrak{s}_j \in \mathfrak{S}_{iter}$ a steepest descent local search is applied. The neighborhood structure for this local search is $\mathcal{N}_{1c,\text{GSS}}$, as introduced in Section 3.

EliteAction($\mathfrak{s}_{ib}$): In every iteration, a short run ($\frac{1}{2} \cdot |O|$ iterations) of a simple TS, which is an extension of the above mentioned local search method, is applied to the iteration-best solution $\mathfrak{s}_{ib}$. The tabu list, whose tenure we have set to 10, ensures that moves cannot be reversed. This algorithmic component is especially useful when the algorithm is applied to JSS instances, in which case it improves the algorithm performance by 3% on average. When applied to OSS instances, this algorithmic component does not contribute to the algorithm performance.

Update($\mathfrak{s}_{ib}, \mathfrak{s}_{rb}, \mathfrak{s}_{bs}$): This function updates solutions $\mathfrak{s}_{rb}$ and $\mathfrak{s}_{bs}$ with the iteration-best solution $\mathfrak{s}_{ib}$. $\mathfrak{s}_{rb}$ is replaced by $\mathfrak{s}_{ib}$, if $C_{\max}(\mathfrak{s}_{ib}) < C_{\max}(\mathfrak{s}_{rb})$. The same holds true for $\mathfrak{s}_{bs}$.

ApplyPheromoneUpdate(cf, bs_update, $\mathcal{T}$, $\mathfrak{s}_{rb}$, $\mathfrak{s}_{bs}$): The typical schedule for updating the pheromone values in $\mathcal{MM}$AS algorithms in the HCF involves three different solutions. These are $\mathfrak{s}_{ib}$, $\mathfrak{s}_{rb}$ and $\mathfrak{s}_{bs}$. Depending on the convergence factor cf, a weight is given to each of these solutions determining their influence on the pheromone update. However, preliminary experiments showed that for OSS problem instances the influence of solution $\mathfrak{s}_{ib}$ must be lower than for JSS instances. The reason is the following. Given a solution $\mathfrak{s}_1$ of an OSS problem instance, a solution $\mathfrak{s}_2$ is obtained by reversing every possible processing order. The makespans of solutions $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are the same. Therefore, if the weight of the iteration-best solution $\mathfrak{s}_{ib}$ is too high, each solution $\mathfrak{s}_1$ competes with the solution $\mathfrak{s}_2$, with reversed processing orders. This slows down the convergence speed of the algorithm considerably. In order to avoid having to fine-tune the use of the iteration-best solution $\mathfrak{s}_{ib}$, depending on the problem instance, we decided not to use the iteration-best solution at all. Therefore, our approach uses only one solution $\mathfrak{s}$ at any time, which is either $\mathfrak{s}_{rb}$ in the case bs_update = FALSE, or $\mathfrak{s}_{bs}$ otherwise, for updating the pheromone values according to the following rule:

$$\tau_{ij} \leftarrow f_{mmas}(\tau_{ij} + \rho \cdot (\delta(o_i, o_j, \mathfrak{s}) - \tau_{ij})), \tag{10}$$

where $\rho \in [0, 1]$ is the evaporation rate. For all our experiments we chose the setting of $\rho = 0.1$. Furthermore,

$$\delta(o_i, o_j, \mathfrak{s}) = \begin{cases} 1 & \text{if } o_i \text{ is to be processed before } o_j \text{ in } \mathfrak{s}, \\ 0 & \text{otherwise,} \end{cases} \tag{11}$$

and

$$f_{mmas}(x) = \begin{cases} \tau_{\min} & \text{if } x < \tau_{\min}, \\ x & \text{if } \tau_{\min} \leqslant x \leqslant \tau_{\max}, \\ \tau_{\max} & \text{if } x > \tau_{\max}. \end{cases} \quad (12)$$

We set the lower bound $\tau_{\min}$ for the pheromone values to 0.001 and the upper bound $\tau_{\max}$ to 0.999. Therefore, after applying the pheromone update we set those pheromone values that exceed the upper bound back to the upper bound value, and for the lower bound respectively.

ComputeConvergenceFactor($\mathcal{T}$): To assess the "extent of being stuck" in an area of the search space, after every iteration we compute the value of a so-called *convergence factor*, *cf*. We compute this value in the following way:

$$cf = 2 \cdot \left(\left(\frac{\sum_{\mathcal{T}_{ij} \in \mathcal{T}} \max\{\tau_{\max} - \tau_{ij}, \tau_{ij} - \tau_{\min}\}}{|\mathcal{T}| \cdot (\tau_{\max} - \tau_{\min})} \right) - 0.5 \right). \quad (13)$$

When the algorithm is initialized (or restarted) with pheromone values all equal to 0.5, *cf* is 0.0 and when all pheromone values are either equal to $\tau_{\min}$ or equal to $\tau_{\max}$, *cf* is 1.0.

5. Experimental Evaluation

We first describe the problem instances that we have selected or generated for the comparison. Then we show experimental results aimed at selecting one of the possible ways of defining the heuristic information, and one of the possible candidate list strategies for restricting the set of operations extending the current partial solution in each construction step. The last part concerns the experimental comparison of ACO_GSS to our adaptation of the TS by Nowicki and Smutnicki [32] to GSS.

5.1. PROBLEM INSTANCES

We decided to apply our algorithm to real GSS, as well as to established OSS and JSS benchmark instances. The only existing GSS instance, named whizzkids97, was introduced in a mathematics competition at the TU Eindhoven, The Netherlands, in 1997 [39]. It consists of 197 operations, 15 machines, and 20 jobs that are subpartitioned into 124 groups. Due to a lack of more GSS instances, we used well-established JSS benchmark instances to generate additional GSS instances. One of the most prominent JSS problem instances is the problem instance ft10, with 10 machines and 10 jobs. It was introduced in [31]. This problem had been open for more than twenty years before the optimality of a solution (with quality 930) was proven by Carlier and Pinson [11]. We chose the classical problem instance ft10, and arbitrarily la38 (15 machines and 15 jobs) from the benchmark set proposed in [27], and abz7 (15 machines and 20 jobs) from the benchmark set provided in [1] to generate new benchmark instances in the following way: For the three problems we refined the job partition into a group partition by subdividing each $J_i = o^i_1 \preceq \cdots \preceq o^i_{|J_i|}$ into b groups of fixed length $g = 1, \ldots, 10$ in the case of ft10, and $g = 1, \ldots, 15$ in the case of the two other problem instances (and possibly one last group of shorter length):

$$\{o^i_1, \ldots, o^i_g\}, \{o^i_{g+1}, \ldots, o^i_{2g}\}, \ldots, \{o^i_{(b-1)g+1}, \ldots, o^i_{|J_i|}\} \quad (b = \lceil |J_i|/g \rceil).$$

This gives us a new benchmark set of 40 GSS instances in the range between open shop scheduling and job shop scheduling.* We denote these instances by the scheme ⟨original_name⟩_⟨group_length⟩. For example, the GSS instance derived from ft10 with group length 3 is denoted by ft10_3.

Additionally, we also tested our algorithm on established JSS and OSS benchmark instances. For the JSS problem we chose another 13 problem instances; among them the set of problem instances that is often called the 10 tough problems. These are abz5 and abz6 (10 jobs, 10 machines), abz7, ..., abz9 (20 jobs, 15 machines), la21, la24 (15 jobs, 10 machines), la25, la27, la29 (20 jobs, 10 machines), la38, la40 (15 jobs, 15 machines), orb08 and orb09 (10 jobs, 10 machines) introduced in [2], and ft20 (20 jobs, 5 machines) introduced in [31]. For the OSS problem, we applied our algorithm to the 10 biggest benchmark instances provided by Taillard [38] (denoted by tai_20x20_∗; 20 jobs, 20 machines), to 8 of the biggest instances provided by Brucker et al. [9] (denoted by j8∗; 8 jobs, 8 machines) and to the 10 biggest instances by Guéret and Prins [23] (denoted by gp10-∗; 10 jobs, 10 machines). Note that the instances by Brucker et al., respectively by Guéret and Prins are more difficult to solve than the Taillard instances. Altogether this makes a sum of 82 problem instances.

5.2. PARAMETER SETTINGS OF ACO_GSS

In the following we experimentally determine the best setting of two parameters of ACO_GSS's solution construction mechanism: (1) The heuristic information that is used for biasing the transition probabilities, and (2) the candidate list strategies that are used to restrict the set of operations for extending the current partial solution. First, we present the experiments that we conducted with the aim of selecting heuristic information to bias the transition probabilities. In order to bias these transition probabilities we use different dispatching rules, i.e., policies for the list scheduler algorithm on which operation to select from the set $\mathcal{O}'_t$ (see Section 2.2) of operations that may be scheduled next. We tested 9 versions of ACO_GSS, corresponding to the 8 different heuristics as shown in Table I (except for the "Random" rule), plus one version that does not use any heuristic information at

* This benchmark set, as well as the other benchmark instances in the GSS input format, is available for download at http://iridia.ulb.ac.be/~cblum/gss/.

all. As an example, for the EST rule we show how to translate it into the heuristic information. In this case the heuristic information is defined by

$$\eta(o_i) \leftarrow \frac{\frac{1}{t_{es}(o_i, \mathfrak{s}^p)+1}}{\sum_{o_k \in \mathcal{O}_t{}'} \frac{1}{t_{es}(o_k, \mathfrak{s}^p)+1}}, \quad \forall o_i \in \mathcal{O}_t{}'. \tag{14}$$

Hereby, 1 is added to the earliest starting times in order to avoid division by 0. For all 9 versions we set $\mathcal{O}_t \leftarrow \mathcal{O}_t{}'$ (i.e., no candidate list strategies are used). We applied the 9 versions of ACO_GSS – each 20 times with a time limit of 180 seconds – to the 10 GSS problem instances derived from the JSS instance ft10. We present a rank-based analysis of the results in Table II(a). The rank of an algorithm version (between 1 and 9) corresponds to its position in the ordered list (decreasing order) of average solution qualities obtained by the 9 versions. The results show that the best average rank is obtained by using the heuristic information based on the EST dispatching rule. Another interesting observation is that the algorithm that is not using any heuristic information at all was only beaten by the algorithms that are using the EST heuristic, respectively the EFT heuristic. All the other heuristics often misled ACO_GSS. A possible reason is that, in those cases where these heuristics do not point the algorithm into the right direction, they are rather harmful, whereas this does not seem to be the case for EFT and EST. With respect to the obtained results, we chose the heuristic information that is based on the EST dispatching rule for the final experimental evaluation of our algorithm.

The second open question after deciding on a version of heuristic information is the choice of the candidate list strategy for restricting the set of operations for extending the current partial solution in each construction step. As outlined in the description of the construction mechanism, there are three different possibilities that are denoted by NR, ND and GT. Instead of testing just these three possibilities, we also tested all possible combinations of these candidate list strategies. A combination of candidate list strategies is achieved when each ant, before it constructs a solution, uniformly chooses one strategy at random from the set of allowed ones. We applied the resulting 7 versions of ACO_GSS each 20 times with a time limit of 180 seconds to the 10 GSS problem instances derived from the JSS instance ft10. The rank-based results are shown in Table II(b). They show that the algorithm version using a combination of ND and NR seems to outperform the other algorithm versions. Therefore, we chose this candidate list strategy for our final experimental evaluation.

5.3. RESULTS AND COMPARISON

We have compared the results obtained by ACO_GSS to our adaptation of the TS approach by Nowicki and Smutnicki [32], which is one of the state-of-the-art algorithms for the JSS problem. Our adaptation of this TS approach is obtained as follows. First, we exchange the original neighborhood structure with our neighborhood structure from Section 3. In this way the algorithm can be applied to all GSS

Table II. Tuning results

Instance	Ranks of the average solution qualities								
	none	EST	EFT	SPT	LPT	LWR	MWR	LTW	MTW
ft10_1	2	1	3	5	9	7	4	6	8
ft10_2	3	2	1	5	9	7	4	6	8
ft10_3	2	1	3	5	7	8	4	6	9
ft10_4	3	1	2	5	7	9	4	6	8
ft10_5	3	1	2	5	7	9	4	6	8
ft10_6	3	1	2	4	6	9	5	7	8
ft10_7	3	1	2	4	6	9	5	7	8
ft10_8	3	1	2	5	6	9	4	7	8
ft10_9	2	1	3	6	5	9	4	7	8
ft10_10	1.5	1.5	3	6	5	9	4	7	8
Average rank	2.55	**1.15**	2.3	5.0	6.7	8.5	4.2	6.5	8.1

(a) Results for 9 versions of ACO_GSS that differ in the dispatching rule they use as heuristic information. The numbers in the table show the rank of the average solution quality obtained per problem instance for each algorithm version. For example, the algorithm version that uses the EFT heuristic was the third-best among the 9 algorithm versions when applied to problem instance ft10_1. The last row gives the average ranks obtained over all problem instances.

Instance	Ranks of the average solution qualities						
	NR	ND	GT	ND, NR	GT, NR	GT, ND	GT, ND, NR
ft10_1	2	6	7	1	4	5	3
ft10_2	5	2	7	4	3	6	1
ft10_3	3	2	7	1	6	4	5
ft10_4	4	3	7	1	5	6	2
ft10_5	6	2	7	1	5	4	3
ft10_6	3	4	7	1	5	7	2
ft10_7	5	2	7	1	6	4	3
ft10_8	5	1	7	2	6	4	3
ft10_9	3.5	3.5	7	3.5	3.5	3.5	3.5
ft10_10	3.5	3.5	7	3.5	3.5	3.5	3.5
Average rank	4.0	2.9	7.0	**1.9**	4.7	4.7	2.9

(b) Results for 7 versions of ACO_GSS that differ in the candidate list strategy that is used for restricting the set of operations that can be used to extend the current partial solution at each construction step. For each algorithm version, the numbers in the table show the rank of the average solution quality obtained per problem instance. For example, the algorithm version that uses the NR strategy is the second-best among all algorithm versions when applied to problem instance ft10_1. The last row gives the average ranks obtained over all problem instances.

instances. We did not implement the cycle detection mechanism. Furthermore, we have made the number of allowed iterations without improvement dependent on the tackled problem instance. This number is given by

$$\left\lfloor |\mathcal{O}| \cdot \left(10 + \left(40 \cdot \frac{1 - (|\mathcal{J}|/|\mathcal{G}|)}{1 - (|\mathcal{J}|/|\mathcal{O}|)}\right)\right)\right\rfloor, \tag{15}$$

and is therefore highest for JSS instances and lowest for OSS instances (decreasing in between). The reason is that the power of the ND algorithm (see Section 2.2) that we use to construct the initial solutions is much higher for OSS instances than for JSS instances. The setting as described above tries to exploit this fact. There is one exception. After a hard restart (which happens when the list of elite solutions is empty) the number of allowed iterations without improvement is twice as high as usual. Finally, we also changed the stopping criteria, in the sense that a CPU time limit is used to stop the algorithm. The resulting TS algorithm performs – with respect to the computation time limits that we applied – about 5 to 10 hard restarts when applied to JSS instances, and about 50 to 100 hard restarts when applied to OSS instances. We henceforth refer to the above explained TS algorithm as TS_GSS.

The results of ACO_GSS in comparison to TS_GSS are shown in Tables III and IV. All the test results were obtained on PCs with AMD Athlon 1100 MHz CPUs under Linux. The format of the result tables is as follows: In the first column we indicate the problem instance. In the second column we give the best objective function value known for the corresponding instance. If this value is denoted in brackets, it means that it is *not* proven to be the optimal solution value. Furthermore, a left arrow (←) indicates that the best known solution value was improved by ACO_GSS. Further, there are two times four columns to specify the results of ACO_GSS and those of TS_GSS. In the first one of these four columns we denote the value of the best solution found in 20 runs of the algorithm. The second one gives the average of the values of the best solutions found in 20 runs. In the third column we indicate the standard deviation of the average given in the second column, and in column 4 we denote the average time that was needed to find the best solutions of the 20 runs. Finally the last column of each table gives the time limit for the algorithms. A value in a column listing the best solution values found is indicated in bold, if the best value found by the other algorithm is worse. In the case of ties we first use the average solution qualities to distinguish among the algorithms, and if this cannot break the tie, we use the average computation times. Furthermore, if the best known solution for an instance was found, the respective value is marked by an asterisk.

The first instance in each section of Table III showing the results for our new GSS benchmark instances is the original JSS instance (ft10, la38, resp. abz7). Then, going down the list, the instances become closer and closer to OSS instances. The last instance in each section is therefore the OSS version of the original JSS benchmark instance. First of all, we tested the statistical significance of the differences

Table III. Results for 41 GSS instances

Instance	Best known	ACO_GSS Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	TS_GSS Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	Time limit
ft10_1	930	*930	938.899	7.608	93.489	***930**	931.899	3.322	66.411	180 s
ft10_2	(872)	**875**	885.6	5.688	103.653	876	880.95	3.634	83.303	"
ft10_3	(827)	835	853.1	8.831	107.022	**828**	840.299	7.567	73.236	"
ft10_4	(782)	799	804	4.291	103.129	**786**	796.2	3.833	81.37	"
ft10_5	(745)	753	761.549	4.639	103.496	***745**	754.35	4.451	90.429	"
ft10_6	(725)	**726**	734.45	5.491	97.808	727	731.7	3.341	89.149	"
ft10_7	(684)	694	700.7	5.722	82.23	***684**	700.75	7.58	100.733	"
ft10_8	(655)	***655**	655.45	0.759	53.917	*655	657.95	1.731	87.132	"
ft10_9	(655)	***655**	655	0	1.375	*655	655	0	12.202	"
ft10_10	(655)	***655**	655	0	0.517	*655	655	0	0.627	"
la38_1	1196	1227	1235.45	4.173	928.019	***1196**	1201.4	1.846	867.177	1800 s
la38_2	(1106)	1120	1144.3	11.304	1099.65	**1109**	1118.42	5.48	999.203	"
la38_3	(1049)	1058	1065.65	3.483	967.941	***1049**	1057.45	5.443	1009.39	"
la38_4	(997)	1019	1029.55	5.306	1128.29	***997**	1014.65	5.677	936.217	"
la38_5	(990)	1006	1018.9	6.866	1156.4	***990**	1001.58	4.426	917.603	"
la38_6	(969)	975	984.299	5.516	958.52	***969**	980.6	4.827	829.38	"
la38_7	(954)	967	978.799	5.425	1113.29	***954**	965.421	5.047	1011.5	"
la38_8	(951)	957	968.75	5.495	1106.96	***951**	959.25	3.753	956.895	"
la38_9	(957)	967	974.649	6.019	1158.81	***957**	968.049	5.185	821.922	"
la38_10	(970)	***970**	984.649	5.896	1072.7	976	982.1	3.537	878.888	"
la38_11	(979)	981	985.85	3.528	1128.67	***979**	984.684	3.972	976.034	"
la38_12	(946)	***946**	951.899	3.275	842.301	948	955.473	4.376	771.169	"
la38_13	(943)	***943**	943	0	40.499	*943	943	0	338.966	"
la38_14	(943)	***943**	943	0	14.335	*943	943	0	218.949	"
la38_15	(943)	***943**	943	0	7.2	*943	943	0	52.284	"
abz7_1	656	674	681.2	3.155	958.764	**666**	668.45	1.47	827.97	1800 s
abz7_2	(641)	*641	645.399	2.891	814.112	***641**	641	0	241.013	"
abz7_3	(612)	*612	612.399	0.882	734.64	***612**	612	0	112.837	"
abz7_4	(609)	*609	609	0	96.472	***609**	609	0	84.519	"
abz7_5	(638)	***638**	638	0	6.916	*638	638.049	0.223	88.302	"
abz7_6	(600)	***600**	600	0	36.448	*600	600	0	121.834	"
abz7_7	(567)	***567**	569	2.752	939.733	*567	569.85	1.598	959.039	"
abz7_8	(577)	***577**	577	0	58.301	*577	577	0	206.838	"
abz7_9	(577)	***577**	577	0	35.159	*577	577	0	404.748	"
abz7_10	(612)	***612**	612	0	14.237	*612	612.6	0.994	814.93	"
abz7_11	(610)	***610**	610	0	4.246	*610	611.75	5.408	221.62	"
abz7_12	(592)	***592**	592	0	4.81	*592	592.399	1.788	75.058	"
abz7_13	(581)	***581**	581	0	9.173	*581	581.1	0.307	256.19	"
abz7_14	(562)	***562**	562	0	8.991	*562	562.149	0.67	131.104	"
abz7_15	(556)	*556	556	0	3.728	***556**	556	0	0.491	"
whizzkids97	469	486	495.149	5.06	1117.55	**475**	482.75	2.953	806.125	1800 s

between the two algorithms by means of a two-sided Wilcoxon rank sum test [13] for every GSS problem instance. Except for the abz7_$*$ instances ($* > 2$) for most other GSS instances we could reject the hypothesis that the two algorithms behave equally, in favor of the hypothesis that they behave differently with a confidence level of 0.95 (p-value < 0.05). The results show that TS_GSS has advantages for more JSS-like instances, whereas ACO_GSS has advantages for more OSS-like

Table IV. Results for existing OSS and JSS benchmark instances

Instance	Best known	ACO_GSS Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	TS_GSS Best	Average	$\sqrt{\sigma^2}$	$\bar{t}$	Time limit
tai_20x20_1	1155	*\textbf{1155}* → ***1155**	1156.85	1.424	197.4	1156	1165.05	4.773	193.977	400 s
tai_20x20_2	1241	**1243**	1247.6	2.37	184.729	1253	1258.15	3.013	148.349	"
tai_20x20_3	1257	***1257**	1257.35	0.587	164.762	1258	1262.6	2.835	121.819	"
tai_20x20_4	1248	***1248**	1248.1	0.307	121.766	*1248	1251.15	2.56	76.846	"
tai_20x20_5	1256	***1256**	1256.35	0.587	196.013	1257	1262.45	3.363	135.449	"
tai_20x20_6	1204	***1204**	1205.15	0.933	195.919	1205	1211.1	4.089	151.525	"
tai_20x20_7	1294	**1295**	1298.2	2.041	246.233	1300	1308.2	3.994	207.826	"
tai_20x20_8	1169	**1173**	1178.85	2.777	191.633	1181	1188.75	5.418	69.906	"
tai_20x20_9	1289	***1289**	1289.05	0.223	133.724	*1289	1293.5	3.9	69.938	"
tai_20x20_10	1241	***1241**	1241.1	0.307	81.916	*1241	1245.05	3.252	91.478	"
j8-per0-1	(1071) ←	***1071**	1075.35	3.842	338.964	1077	1087.9	5.23	372.327	640 s
j8-per0-2	(1062) ←	***1062**	1072.85	6.054	367.7	1073	1091.1	9.095	336.37	"
j8-per10-0	(1033) ←	***1033**	1046.15	4.704	300.803	1052	1061.45	6.924	300.024	"
j8-per10-1	(1017) ←	***1017**	1024.7	3.262	301.435	1025	1038.6	6.064	223.585	"
j8-per10-2	(1020) ←	***1020**	1027.9	4.789	292.358	1021	1043.95	8.159	266.27	"
j8-per20-0	1000	**1003**	1010.1	2.826	301.759	1011	1017.25	3.668	291.589	"
j8-per20-1	1000	***1000**	1000	0	50.613	*1000	1000.2	0.695	230.703	"
j8-per20-2	(1001) ←	***1001**	1007.55	3.136	256.897	1007	1018.05	5.031	233.673	"
gp10-01	(1108) ←	***1108**	1112.75	3.416	531.083	1118	1139.6	9.794	485.971	1000 s
gp10-02	(1101) ←	***1101**	1113.65	6.523	347.108	1120	1135.75	12.539	484.678	"
gp10-03	(1096) ←	***1096**	1102.5	3.203	561.317	1121	1132.05	6.286	472.247	"
gp10-04	(1083) ←	***1083**	1092	3.699	492.382	1100	1113.7	8.885	450.235	"
gp10-05	(1091) ←	***1091**	1096.2	3.994	486.612	1104	1121.65	9.365	426.275	"
gp10-06	(1071) ←	***1071**	1077.35	10.059	613.248	1115	1132.5	11.274	579.763	"
gp10-07	(1081) ←	***1081**	1082.8	3.286	692.913	1089	1120.9	15.66	459.557	"
gp10-08	(1096) ←	***1096**	1102	3.906	537.129	1115	1134.5	11.362	425.141	"
gp10-09	(1121)	**1124**	1129.7	3.419	608.052	1132	1150.45	9.11	467.763	"
gp10-10	(1092) ←	***1092**	1095.15	2.368	543.205	1123	1144.35	9.258	456.75	"

(a) Results for 28 of the largest OSS benchmark instances

Table IV. (Continued)

Instance	Best known	ACO_GSS Best	ACO_GSS Average	ACO_GSS $\sqrt{\sigma^2}$	ACO_GSS $\bar{t}$	TS_GSS Best	TS_GSS Average	TS_GSS $\sqrt{\sigma^2}$	TS_GSS $\bar{t}$	Time limit
abz7	656	674	681.2	3.155	958.764	**666**	668.45	1.47	827.97	1800 s
abz8	(669)	689	697.049	3.235	1093.25	**673**	679.95	3.17	794.02	"
abz9	(679)	702	709.35	4.158	1059.21	**688**	692.2	2.607	632.092	"
la21	1046	1047	1053.25	3.507	460.265	**1047**	1049.25	2.048	368.105	900 s
la24	935	944	948.1	3.385	363.611	**939**	942.299	1.38	311.956	"
la25	977	*977	981.45	2.981	894.611	***977**	977.299	0.47	676.827	1800 s
la27	1235	1243	1255.5	5.898	1031.74	***1235**	1241.15	3.688	895.972	"
la29	(1152)	1168	1186.75	8.149	1084.98	**1164**	1168.1	2.174	822.03	"
la38	1196	1227	1235.45	4.173	928.019	***1196**	1201.4	1.846	867.177	"
la40	1222	1228	1234.55	5.915	1031.1	**1224**	1228.35	2.518	711.315	"
ft10	930	*930	938.899	7.608	93.489	***930**	931.899	3.322	66.411	180 s
ft20	1165	*1165	1168.55	5.114	88.283	***1165**	1165	0	22.501	"
orb08	899	*899	914.649	6.869	88.263	***899**	910.75	6.331	70.376	"
orb09	934	*934	935.149	2.924	80.496	***934**	934	0	27.816	"
abz5	1234	*1234	1237.2	1.361	34.177	***1234**	1236.9	1.372	54.466	"
abz6	943	947	947.799	0.41	15.369	***943**	943.7	0.978	61.444	"

(b) Results for 16 JSS benchmark instances

instances. This is not just true for the solution qualities obtained, but also for the average computation times needed. ACO_GSS finds the best solutions of a run more quickly for more OSS-like instances, and vice versa. Furthermore, the same observation can be made for the standard deviation of the average solution qualities obtained. TS_GSS does not just find better solutions for JSS-like instances, but is usually also characterized by a lower standard deviation of the best solution values found over 20 runs. In turn, the same holds for ACO_GSS for more OSS-like instances. The difficult whizzkids97 instance is on 197 operations and 124 groups. This means that it is quite close to a JSS instance. Consequently, TS_GSS performs better. Both approaches find the optimal solution value (930) for the ft10 JSS instance, and TS_GSS also finds the optimal solution (1196) for the difficult la38 JSS instance.

Table IV(a) shows the results obtained by the two approaches for the biggest OSS benchmark instances that exist (see Section 5.1). The results confirm the impression that was given by the results for the GSS instances. For OSS instances, ACO_GSS is by all means clearly superior to TS_GSS. This is more obvious for the benchmark instances by Brucker et al. and those by Guéret and Prins than for the benchmark instances by Taillard, which are relatively easy to solve. ACO_GSS is usually better in the best solution values found, in average solution qualities obtained, in standard deviation of the best solution values obtained, and generally also in the average time needed to find the best solutions. The results of TS_GSS confirm that algorithms that reach state-of-the-art performance for a certain problem cannot, in general, be adapted to other problems in such a way that they remain highly functional.

ACO_GSS is able to improve the best known solution values for 6 of the 8 instances by Brucker et al., and for 9 of the 10 instances by Guéret and Prins.* The only algorithm that was applied to these instances before is the EC algorithm by Prins [34]. That ACO_GSS beats this algorithm so clearly is remarkable, because in contrast to this EC algorithm ACO_GSS is not specialized to solve OSS instances.

Finally, Table IV(b) shows the results of ACO_GSS in comparison to TS_GSS for the "10 tough problems" from the JSS literature, as well as 6 easier (smaller) JSS instances. Observing the results, it becomes clear that TS_GSS in general has obvious advantages over the ACO approach when applied to JSS problem instances. This especially holds for the bigger problem instances. This result is not surprising as the JSS version of TS_GSS is one of the state-of-the-art algorithms for JSS. However, ACO_GSS is the first ACO approach that obtains an acceptable performance for JSS instances. This is documented by the fact that it is the first ACO algorithm that finds the solution of the ft10 instance by Fisher and Thompson, which for a long time was the ultimate challenge for JSS algorithms.

6. Conclusions

We have proposed an ant colony optimization approach to tackle the broad class of group shop scheduling problem instances. Our approach is a $\mathcal{MAX}$-$\mathcal{MIN}$ ant system in the hyper-cube framework. It probabilistically constructs solutions using the ND algorithm. Furthermore, it employs black-box local search procedures for improving the constructed solutions. These local search procedures are based on a new neighborhood for the group shop scheduling problem. This neighborhood is an adaptation of the successful neighborhood derived by Nowicki and Smutnicki for the JSS problem. After fine-tuning the construction mechanism of our ant colony optimization approach, we did an experimental evaluation of our new method and compared the results to an adaptation of the successful tabu search approach by Nowicki and Smutnicki to GSS. The results showed that ACO_GSS is especially suited to the application in OSS instances. We were able to improve the best known solution value for 15 of the 28 tested OSS instances. This is remarkable as our algorithm is not specialized in solving OSS problem instances. Also the performance of our ant colony optimization approach is acceptable for JSS problem instances.

* Note that a new state-of-the-art algorithm for OSS has recently been accepted for publication (see [6]). This algorithm further improves 10 of the 15 best known solutions that are improved by our algorithm.

In particular, it is the first ant colony optimization approach that can solve the ft10 instance by Fischer and Thompson.

References

1. Adams, J., Balas, E. and Zawack, D.: The shifting bottleneck procedure for job shop scheduling, *Management Sci.* **34**(3) (1988), 391–401.
2. Applegate, D. and Cook, W.: A computational study of the job-shop scheduling problem, *ORSA J. Comput.* **3** (1991), 149–156.
3. Balas, E. and Vazacopoulos, A.: Guided local search with shifting bottleneck for job shop scheduling, *Management Sci.* **44**(2) (1998), 262–275.
4. Blażewicz, J., Domschke, W. and Pesch, E.: The job shop scheduling problem: Conventional and new solution techniques, *European J. Oper. Res.* **93** (1996), 1–33.
5. Blum, C.: AGO applied to group shop scheduling: A case study on intensification and diversification, in M. Dorigo, G. Di Caro and M. Sampels (eds), *Proceedings of ANTS 2002 – From Ant Colonies to Artificial Ants: Third International Workshop on Ant Algorithms*, Lecture Notes in Comput. Sci. 2463, Springer Verlag, Berlin, 2002, pp. 14–27.
6. Blum, C.: Beam-ACO – Hybridizing ant colony optimization with beam search: An application to open shop scheduling, *Comput. Oper. Res.* (2004), in press.
7. Blum, C. and Dorigo, M.: The hyper-cube framework for ant colony optimization, *IEEE Trans. Systems Man Cybernet. – Part B* **34**(2) (2004), 1161–1172.
8. Brinkkötter, W. and Brucker, P.: Solving open benchmark problems for the job shop problem, *J. Scheduling* 4 (2001), 53–64.
9. Brucker, P., Hurink, J., Jurisch, B. and Wostmann, B.: A branch & bound algorithm for the open-shop problem, *Discrete Appl. Math.* **76** (1997), 43–59.
10. Brucker, P., Jurisch, B. and Sievers, B.: A branch and bound algorithm for the job-shop scheduling problem, *Discrete Appl. Math.* **49** (1994), 109–127.
11. Carlier, J. and Pinson, E.: An algorithm for solving the job-shop problem, *Management Sci.* **35**(2) (1989), 164–176.
12. Colorni, A., Dorigo, M., Maniezzo, V. and Trubian, M.: Ant system for job-shop scheduling, *JORBEL – Belgian J. Oper. Res., Statist. Comput. Sci.* **34**(1) (1994), 39–53.
13. Conover, W.: *Practical Nonparametric Statistics*, Wiley Series in Probability and Statistics, Wiley, New York, NY, 1999.
14. Dell'Amico, M. and Trubian, M.: Applying tabu search to the job-shop scheduling problem, *Ann. Oper. Res.* **41** (1993), 231–252.
15. den Besten, M. L., Stützle, T. and Dorigo, M.: Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem, in E. J. W. Boers, J. Gottlieb, P. L. Lanzi, R. E. Smith, S. Cagnoni, E. Hart, G. R. Raidl and H. Tijink (eds), *Applications of Evolutionary Computing: Proceedings of EvoWorkshops 2001*, Lecture Notes in Comput. Sci. 2037, Springer Verlag, Berlin, 2001, pp. 441–452.
16. Dorigo, M. and Di Caro, G.: The ant colony optimization meta-heuristic, in D. Corne, M. Dorigo and F. Glover (eds), *New Ideas in Optimization*, McGraw-Hill, London, 1999, pp. 11–32.
17. Dorigo, M., Maniezzo, V. and Colorni, A.: Ant system: Optimization by a colony of cooperating agents, *IEEE Trans. Systems Man Cybernet. – Part B* **26**(1) (1996), 29–41.
18. Dorndorf, U. and Pesch, E.: Evolution based learning in a job shop scheduling environment, *Comput. Oper. Res.* **22** (1995), 25–40.
19. Dorndorf, U., Pesch, E. and Phan-Huy, T.: Solving the open shop scheduling problem, *J. Scheduling* **4**(3) (2001), 157–174.

20. Dorndorf, U., Pesch, E. and Phan-Huy, T.: Constraint propagation and problem decomposition: A preprocessing procedure for the job shop problem, *Ann. Oper. Res.* **115**(1) (2002), 125–145.
21. Fang, H.-L., Ross, P. and Corne, D.: A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems, in *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA '93)*, Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 375–382.
22. Giffler, B. and Thompson, G. L.: Algorithms for solving production scheduling problems, *Oper. Res.* **8** (1960), 487–503.
23. Guéret, C. and Prins, C.: A new lower bound for the open-shop problem, *Ann. Oper. Res.* **92** (1999), 165–183.
24. Haupt, R.: A survey of priority rule-based scheduling, *OR Spektrum* **11** (1989), 3–16.
25. Ikeda, K. and Kobayashi, S.: GA based on the UV-structure hypothesis and its application to JSP, in *Proceedings of PPSN-VI, Sixth International Conference on Parallel Problem Solving from Nature*, Springer-Verlag, Berlin, 2000, pp. 273–282.
26. Jain, A. and Meeran, S.: Deterministic job-shop scheduling; past, present and future, *European J. Oper. Res.* **113**(2) (1999).
27. Lawrence, S.: Resource constraint project scheduling: An experimental investigation of heuristic scheduling techniques (Supplement), Technical Report, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, USA, 1984.
28. Liaw, C.-F.: A tabu search algorithm for the open shop scheduling problem, *Comput. Oper. Res.* **26** (1999), 109–126.
29. Liaw, C.-F.: A hybrid genetic algorithm for the open shop scheduling problem, *European J. Oper. Res.* **124** (2000), 28–42.
30. Merkle, D., Middendorf, M. and Schmeck, H.: Ant colony optimization for resource-constrained project scheduling, *IEEE Trans. Evol. Comput.* **6**(4) (2002), 333–346.
31. Muth, J. F. and Thompson, G. L.: *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs, NJ, 1963.
32. Nowicki, E. and Smutnicki, C.: A fast taboo search algorithm for the job-shop problem, *Management Sci.* **42**(2) (1996), 797–813.
33. Pfahringer, B.: A multi-agent approach to open shop scheduling: Adapting the ant-Q formalism, Technical Report TR-96-09, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien, Austria, 1996.
34. Prins, C.: Competitive genetic algorithms for the open-shop scheduling problem, *Math. Methods Oper. Res.* **52**(3) (2000), 389–411.
35. Roy, B. and Sussmann, B.: Les problémes d'ordonnancement avec constraints dijonctives, Technical Report Note DS 9 bis, SEMA, Paris, France, 1964.
36. Stützle, T.: An ant approach to the flow shop problem, in *Fifth European Congress on Intelligent Techniques and Soft Computing, EUFIT'98*, 1998, pp. 1560–1564.
37. Stützle, T. and Hoos, H. H.: $\mathcal{MAX}$-$\mathcal{MIN}$ ant system, *Future Generation Computer Systems* **16**(8) (2000), 889–914.
38. Taillard, É. D.: Benchmarks for basic scheduling problems, *European J. Oper. Res.* **64** (1993), 278–285.
39. Whizzkids: http://www.win.tue.nl/whizzkids/1997.
40. Yamada, T. and Nakano, R.: Job-shop scheduling by simulated annealing combined with deterministic local search, in *Meta-Heuristics: Theory & Applications*, Kluwer Acad. Publ., Boston, MA, 1996, pp. 237–248.