# Genetic Local Search for Job Shop Scheduling Problem

A. MORAGLIO (*), H.M.M. TEN EIKELDER(**), R. TADEI (*)

(*) Politecnico di Torino, Dipartimento di Automatica e Informatica, Corso Duca degli Abruzzi 24, 10129 Torino, Italy, Tel. +39-11-5647032, fax +39-11-5647099, e-mail: a.moraglio@studenti.to.it, tadei@polito.it

(**) Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, Tel. +31-40-2475153, fax +31-40-2451733, e-mail: h.m.m.t.eikelder@tue.nl

## Abstract

The Job Shop Scheduling Problem is a strongly NP-hard problem of combinatorial optimisation and one of the best-known machine scheduling problem. Taboo Search is an effective local search algorithm for the job shop scheduling problem, but the quality of the best solution found depends on the initial solution. To overcome this problem we present a new approach that uses a population of Taboo Search runs in a Genetic Algorithm framework: GAs localise good areas of the solution space so that TS can start its search with promising initial solutions. The peculiarity of the Genetic Algorithm we propose consists in a natural representation which covers all and only the feasible solution space and guarantees the transmission of meaningful characteristics. The results show that this method outperforms many others producing good quality solutions in less time.

*Keywords*: Job Shop, Taboo Search, Genetic Algorithms, Hybrid Optimisation

# 1 Introduction

The Job Shop Scheduling Problem (JSSP) is considered as a particularly hard combinatorial optimisation problem. The problem has been studied by many authors and several algorithms have been proposed. Only very special cases of the problem can be solved in polynomial time, but their immediate generalisations are NP-hard [9].

The form of the JSSP may be roughly sketched as follows: we are given a set of jobs and a set of machines. Each machine can handle at most one job at a time. Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The purpose is to find a schedule, that is an allocation of the operations to time intervals on the machines, that has minimum length.

Taboo Search (TS) is a local search method designed to find a near-optimal solution of combinatorial optimisation problems [6]. One peculiarity of TS is a short term memory used to keep track of recent solutions which are considered forbidden (taboo), thus allowing the search to escape from local optima.

TS has revealed to be an effective local search algorithm for the Job Shop Scheduling Problem [13, 16]. However, the best solution found by TS may depend on the initial solution used. This is essentially due to the fact that, like any other local search technique, TS starts its search from a single solution, which may lead the search to a dead-end despite the presence of the taboo mechanism, which would prevent it. This happens especially when TS is applied to particularly hard optimisation problem like JSSP.

Genetic Algorithms (GAs) are stochastic global search methods that mimic the natural biological evolution [7]. GAs operate on a population of potential solutions applying the principle of survival of the fittest to produce (hopefully) better and better approximations to a solution.

Simple GAs are difficult to apply directly and successfully into many difficult-to-solve optimisation problems. Various non-standard implementations have been created for particular problems in which genetic algorithms are used as meta-heuristics [3, 10]. In this new perspective, Genetic Algorithms are very effective at performing global search (in probability) and provide a great flexibility to hybridise with domain-dependent heuristics to make an efficient implementation for a specific problem.

Several authors have proposed variants of local search algorithms, using ideas from population genetics [1, 11, 19]. Because of the complementary properties of genetic algorithms and conventional heuristics, the hybrid approach often outperforms either method operating alone.

Previous studies on ordering problems as the travelling salesman problem (TSP) have proven that a natural representation is the key-issue for the success of a GA approach [17].

The JSSP is mainly characterised as being a highly constrained ordering problem. Therefore, both aspects have to be considered in order to figure out a natural GAs representation. GA representations for JSSP can be found in [4, 8, 12].

In section 2, we propose a representation that, coupled with a particular class of recombination operators, guarantees the genetic search to represent all and only the feasible solutions and that guarantees the transmission of meaningful characteristics to the offspring solutions. The definition of a class of recombination operators and the choice of an operator showing interesting properties follow.

In section 3, we propose a genetic local search algorithm (GTS) consisting of a basic genetic algorithm with the addition of a taboo search optimisation phase applied to every new individual created.

In section 4, computational experiments show that the combination of GAs and TS performs better than the TS alone. Moreover, a wide comparison of GTS with a variety of algorithms for JSSP on a set of well-known instances of small and medium sizes shows that GTS is very well positioned. Finally, a specific comparison of GTS with a similar approach combining Genetic Algorithms and Simulated Annealing (SAGen) [8] on a set of large size instances shows that GTS outperforms SAGen both with respect to computational time and solutions quality.

# 2 A Genetic Algorithm for JSSP

### *2.1 Appeal to feasibility*

In order to apply GAs to a particular problem we have to encode a generic solution of the problem into a chromosome. How to encode a solution is a key-issue for the success of GAs [3]. Basic GAs use binary encoding of individuals on fixed-length strings. Such a representation is not naturally suited for ordering problems such as the Travelling Salesman Problem and the JSSP, because no direct and efficient way has been found to map all possible solutions into binary strings [17].

The main difficulty in choosing a proper representation for highly constrained combinatorial optimisation problems such as JSSP is dealing with the infeasibility of the solutions produced during the evolutionary process. This problem is typically addressed by modifying the breeding operators, associating them with repair methods, or providing penalties on infeasible solutions in the fitness function, or discarding infeasible solutions when created. However, the use of penalty functions or a rejecting strategy is inefficient for JSSP because the space of feasible schedules is very small compared to the space of

possible schedules, therefore the GA will waste most of its time producing and/or processing infeasible solutions. Repairing techniques are a better choice for many combinatorial optimisation problems since they are easy to apply and surpass strategies such as rejecting strategies and penalising strategies [14]. However, whereas it is possible, the most efficient and direct method remains to embed constraints in the coding of individuals. Thus, a very important issue in building a genetic algorithm for JSSP is to devise an appropriate representation of solutions together with a problem-specific genetic operator so that all chromosomes generated in either the initial phase or the evolutionary process will produce feasible schedules. This is a crucial phase that affects all the subsequent steps of GAs.

In this paper we propose a representation and a particular class of recombination operators that together guarantee the genetic search to cover all the space of feasible solutions, to represent only feasible solutions, and the transmission of meaningful characteristics to the offspring solutions.

Let us first spend few words on the nature of infeasibility. On the basis of what kind of solution representation for the JSSP we consider, two different causes of infeasibility may occur:

- schedules non-respecting all job precedence constraints
- solutions with cycles

Because of the existence of the precedence constraints of operations on jobs, if we consider a solution representation which doesn't presuppose a fixed order of operations on jobs, but rather which can freely dispose operations both on machines and jobs, then mismatches between the order of operations encoded in a generic chromosome and the prescribed order of operations on jobs may arise. Therefore this is a first cause of infeasibility.

In a schedule, two generic operations are allowed to be either processed in parallel (there is no precedence among them) or processed sequentially (in this case, one precedes the other one). What is not possible is that one operation both precedes and follows the other one. If we consider a representation of solutions which allow to encode precedence conflicts between operations like the one just mentioned (i.e. cycling solutions), then we encounter the second cause of infeasibility.

In order to avoid both kinds of infeasibility in our GA, we introduce a class of recombination operators that solves the problem with job constraints and a representation that solves the problem concerning cycling solutions. More in detail, we will see that only solutions without cycles can be represented, thus eliminating the cycling problem. Unfortunately the schedules so represented do not necessarily respect job precedence constraints. However, to manage this second kind of unfeasibility, it suffices initialising the evolutionary process with a population of schedules respecting all job precedence

constraints and applying recombination operators that leave the job precedence constraints invariant.

### *2.2 Representation*

In order to apply the GA framework, we need to define an encoding method to map the search space of all possible solutions into a set of finite chromosomes.

In the sequel we introduce the representation we will use. First, we will show, by means of an example, the relationship among a problem instance, represented by its disjunctive graph, a particular solution for that instance, represented by its solution graph, and our string coding for that solution. Afterwards, we will present definitions and theorems that assure the validity of the representation proposed.

*Problem, Solution and Encoding*

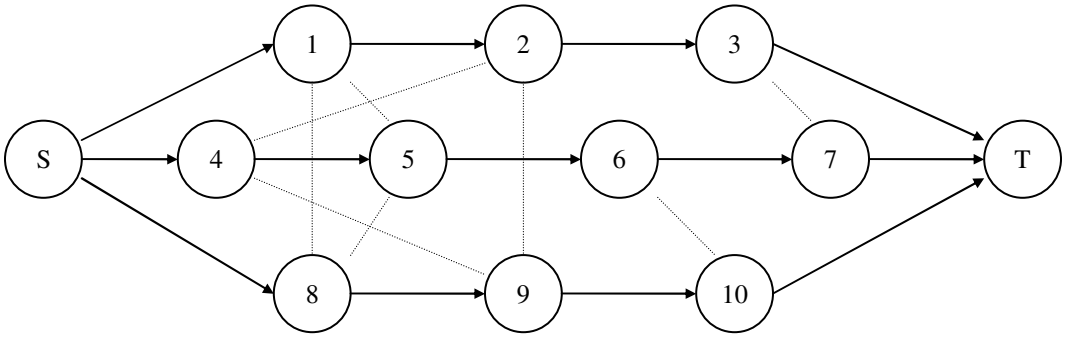


**Figure 1** – Disjunctive Graph (Elements of *A* are indicated by arrows and elements of *E* are indicated by dashed lines.)

The job shop scheduling problem can be represented with a disjunctive graph [15]. A disjunctive graph $G=(N, A, E)$ is defined as follows: $N$ is the set of nodes representing all operations, $A$ is the set of arcs connecting consecutive operations of the same job, and $E$ is the set of disjunctive arcs connecting operations to be processed by the same machine. A disjunctive arc can be settled by either of its two possible orientations. The construction of a schedule will settle the orientations of all disjunctive arcs so as to determine the sequence of operations on the same machine. Once a sequence is determined for a machine, the disjunctive arcs connecting operations to be processed by the machine will be replaced by the oriented precedence arrow, or conjunctive arc. The set of disjunctive arcs $E$ can be decomposed into cliques, one for each machine. The processing time for each operation can be seen as a weight attached to the corresponding nodes. The JSSP is equivalent to finding the order of the operations on each machine, that

is, to settle the orientation of the disjunctive arcs such that the resulting solution graph is acyclic (there are no precedence conflicts between operations) and the length of the longest weighted path between the starting and terminal nodes is minimal. This length determines the makespan.

Figure 1 illustrates the disjunctive graph for a three-job four-machine instance. The nodes of $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ correspond to operations. Nodes $S$ and $T$ are two special nodes, starting node ($S$) and terminal node ($T$), representing the beginning and the end of the schedule, respectively. The conjunctive arcs (arrows) of $A = \{(1, 2), (2, 3), (4, 5), (5, 6), (6, 7), (8, 9), (9, 10)\}$ correspond to precedence constraints on operations on same jobs. The disjunctive arcs (dashed lines) of $E_1 = \{(1, 5), (1, 8), (5, 8)\}$ concern operations to be performed on machine 1, disjunctive arcs $E_2 = \{(2, 4), (2, 9), (4, 9)\}$ concern operations to be performed on machine 2, disjunctive arcs $E_3 = \{(3, 7)\}$ concern operations to be performed on machine 3, and disjunctive arcs $E_4 = \{(6, 10)\}$ concern operations to be performed on machine 4.

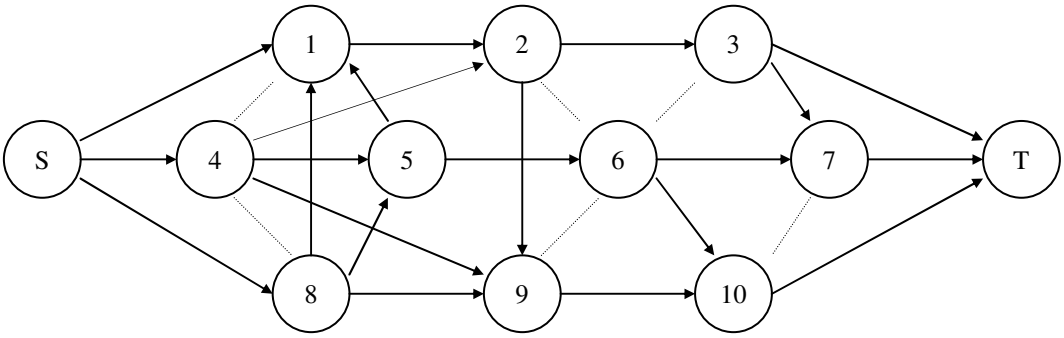


**Figure 2** – Solution Graph (Sequential operations are connected by arrows, parallel operations are connected by dashed lines)

Figure 2 illustrates the solution graph representing a feasible solution to the given instance of the problem. It has been derived from the disjunctive graph described above by settling an orientation of all the disjunctive arcs having taken care to avoid the creation of cycles. In the solution graph, arrows correspond to precedence constraints among operations on jobs or machines. Dashed lines indicate that two operations don't have any precedence constraints (in principle they could be processed in parallel without violating any precedence constraints. In fact their actual parallel processing depends only on the processing time of operations). The sequences of operations on jobs depend only on the instance of the problem and not on the particular solution. In our example they are:

Job1: Op1, Op2, Op3

Job2: Op4, Op5, Op6, Op7
Job3: Op8, Op9, Op10

On the contrary, the sequences of operations on machines also depend on the particular solution to the given problem. In our example they are:

Mac1: Op8, Op5, Op1
Mac2: Op4, Op2, Op9
Mac3: Op3, Op7
Mac4: Op6, Op10

Let us see things under a different perspective, emphasising the precedence order relationship among operations. The disjunctive graph of Figure 1 represents a particular instance of JSSP. We can see it as a partial order relationship among operations. The solution graph shown in Figure 2 represents a specific solution of the above JSSP instance. We can see it still as a partial order relationship among operations, even if more constrained when compared to the relationship associated with the disjunctive graph. We can now force a complete order by imposing further precedence constraints so that obtaining a linear sequence of operations, the string shown in Figure 3, which is the encoding of a solution we will use.

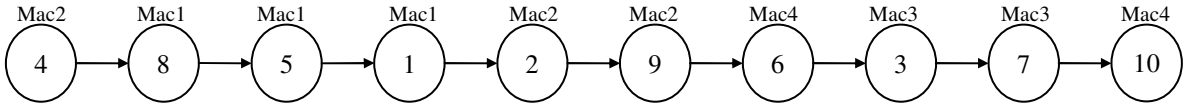


**Figure 3** – String Representation (Complete precedence order among all operations)

In the string is present all information we need to decode it into an actual schedule. Since we know a priori (from the problem instance) the machine which a given operation belongs to, the sequence of operations on each machine is easily determinable from the string. The idea is to scan the string from left to right, extract all the operations of a given machine and sequencing them keeping the same order. Considering again our example, if we apply the decoding procedure just described to the string of Figure 3, it is easy to see that we obtain exactly the same sequences of operations on machines as reported before, the same sequences we have extracted from the solution graph.
A peculiarity of the string representation is that it doesn't admit cyclic solutions. It is therefore not subject to the second kind of infeasibility we have discussed in section 2.1. However, we can notice that a string codifies both information about the solution it represents (precedence constraints on machines) and information about the instance of the problem (precedence constraints on jobs). This implies that a generic string may represent

a solution which does not respect the precedence constraints on jobs, therefore we still have to deal with this kind of infeasibility, that is the first kind discussed in section 2.1.

*Formal Definition of String and Coding/Decoding Theorems*

In the following we give the formal definition of string representation. Then, in order to show that the string representation is a valid encoding for schedules, we formulate two theorems.

**Definition 1.** String Representation.

Let us consider three finite sets, a set $J$ of jobs, a set $M$ of machines and a set $O$ of operations. For each operation $a$ there is a job $j(a)$ in $J$ to which it belongs, a machine $m(a)$ in $M$ on which it must be processed and a processing time $d(a)$. Furthermore for each operation $a$ its successor in the job is given by $sj(a)$, except for the last operation in a job. The representation of a solution is a string consisting of a permutation of all operations in $O$, i.e. an element of the set:

$StrRep = \{ s \in O^n \mid n = |O| \text{ and } \forall\, i, j \text{ with } 1 \le i < j \le n: s(i) \neq s(j) \}$

Now we can define legal strings. Formal for $s$ in $StrRep$:

$Legal(s) = \forall\, a, sj(a) \in O: a \sim\!\prec sj(a)$

where $a \sim\!\prec b$ means: $a$ occurs before $b$ in the string $s$.

**Theorem 1**. (Feasible Solution → Legal String)

Every feasible solution can be represented by a legal string. More than one legal string corresponding to the same feasible solution may exist.

*Proof.*

Every feasible solution can be represented by an acyclic solution graph, say $C$.

Every acyclic solution graph $S$ can be transformed in a legal string by means of the following construction procedure:

1. Set $S$ as the current graph $C$
2. Calculate the transitive closure graph $TC$ of the current graph $C$
3. WHILE the transitive closure $TC$ doesn't define a total order in $O$ DO
    4. Select two nodes in $O$ still not linked by an arc in $TC$
    5. Link them by a directed arc obtaining a new acyclic graph that becomes the new current graph $C$
    6. Calculate the transitive closure graph $TC$ of the current graph $C$
7. Convert the transitive closure graph $TC$ in its corresponding string $Str$

The previous procedure:

- always produces a total order in $O$ and never a cyclic graph. Therefore the conversion of $TC$ in $Str$ in step 7 is immediate
- is non-deterministic in step 4 and 5 and consequently it may produce different strings starting from the same acyclic solution graph $S$
- always produces a legal string since the initial solution graph $S$ is still a sub-graph of the final transitive closure graph $TC$

■

**Theorem 2**. (Legal String → Feasible Solution)
Every legal string corresponds exactly to one feasible solution.
*Proof.*
A generic legal string $Str$ can be interpreted as a complete order relationship among operations and consequently can be associated with an acyclic graph $TC$.
Let us consider the set of (directed) arcs $A$ and the set of (undirected) edges $E$ defined in the disjunctive graph. By eliminating from $TC$ every arc not in $A \cup E$, we obtain a new graph $S$ representing the solution.
Moreover since arcs of the form $[a, sj(a)]$ are present in the graph $TC$ and these arcs are not removed in the elimination process, the resulting solution graph $S$ has the correct job arcs, i.e. it corresponds to a feasible solution.

■

### *2.3 Recombination*

In order to cope with the unfeasibility regarding job precedence constraints we propose the following requirement on the recombination operators that guarantees both the respect of job constraints and the transmission of meaningful characteristics.

**Definition 2.** Feasibility Requirement for Recombination.
We say that a generic recombination operator for the string representation is feasible, if for every generic pair of operations $a$ and $b$ we have $a \sim\!\!< b$ in both parent strings then also $a \sim\!\!< b$ must hold in the child strings produced by its application to the parent strings.

**Theorem 3**. (Legal String + Legal String → Legal String)
By recombining legal strings following the feasibility requirement for recombination, we still obtain a legal string.
*Proof.*

Let *s* and *t* be two legal parent strings. Let offspring *u* be obtained by a recombination that respects the feasibility requirement. By definition we have to show that $a \sim < sj(a)$ for all operations *a* in string *u*. Since *s* and *t* are legal strings, this property holds for *s* and *t*. From the feasibility requirement we immediately conclude that also $a \sim < sj(a)$ for all operations *a* in string *u*.
■

In the following we propose a recombination operator that respects the feasibility requirement.

**Definition 3.** Recombination Operator.
Let *SEQ* be a vector of *n* elements randomly chosen in the set $P=\{1, 2, 3, 4\}$, where $n=|O|$. Each number in *P* denotes a pointer. There is one pointer for each extremity of the parent strings. Each pointer is allowed to move in a given direction (see also Figure 4). The pointers are moved in turn according to their order of appearance in the SEQ vector. The following procedure illustrates how to produce a single offspring string from two parent strings:

1. Initialise the left scan pointers *PT1* and *PT2* at the beginning (on the left side) of parent strings *PAR1* and *PAR2*. Initialise the right scan pointers *PT3* and *PT4* at the end (on the right side) of the parent strings *PAR1* and *PAR2*. Let the result (son) be a string consisting of *n* blanks. Initialise the left write pointer *P1* at the beginning of the result string and initialise the right write pointer *P2* at the end of the result string. Set all operations as unmarked.
2. Consider the first number appearing in the sequence *SEQ*
3. Slide the corresponding scan pointer to the first unmarked operation (left scan pointers slide from left to right, right scan pointers slide from right to left)
4. If the pointer in step 3 was a left pointer, copy the current operation at the left write pointer and increase that pointer by 1. Otherwise copy the current operation at the right write pointer and decrease it by 1. Mark the current operation in the parents as already processed
5. Take out the number at the beginning of *SEQ*
6. If *SEQ* is empty then stop otherwise go to step 2

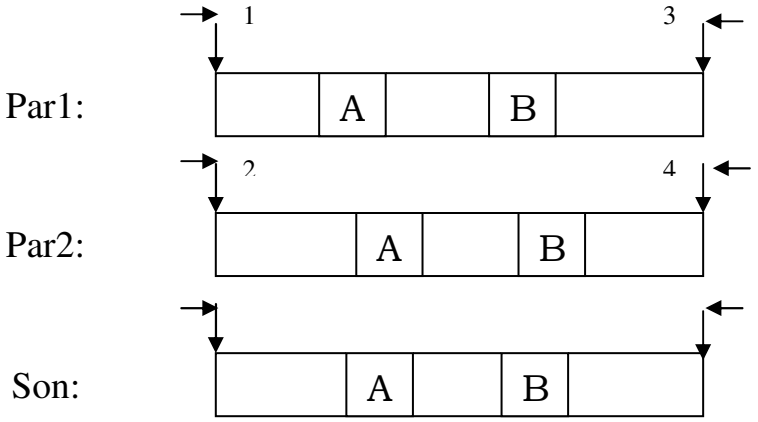


**Figure 4 –** Common order preserving recombination

**Theorem 4**. Validity of Recombination Operator.
The recombination operator defined above respects the feasibility requirement.
*Proof.*
Let us consider the two parent strings in Figure 4. To transmit a generic operation from a parent to the son, that operation must be reached by one of the four scan pointers (indicated in Figure 4 with numbers from 1 to 4). Therefore, to transmit operations *A* and *B* both must be reached by a pointer, one for each. A pair of pointers (*x*, *y*) defines a way to transmit operations *A* and *B* by means of the following procedure:

- First, the *x* pointer slides in its prescribed direction until it reaches operation *A* or operation *B* and transmits it to the son
- Then, the *y* pointer slides in its prescribed direction until it reaches the operation which among *A* and *B* is not yet assigned to the son

In Table 1 all possible pairs of pointers are grouped in four classes of equivalence following two lines of symmetry.

| | **Same Parent** | **Different Parent** |
|---|---|---|
| **Same Side** | (1, 1) (2, 2) (3, 3) (4, 4) | (1, 2) (2, 1) (3, 4) (4, 3) |
| **Different Side** | (1, 3) (3, 1) (2, 4) (4, 2) | (1, 4) (4, 1) (2, 3) (3, 2) |

**Table 1 –** Symmetries of the recombination

Let us consider only one pair for every class, since the other pair in the same class will produce the same result:

- Case (1, 1). The pointer 1 gets first *A* and puts it in the son. The same pointer then gets *B* and puts it in the son more to the right respect to *A*. This is because the pointer used in the son slides in the same direction as pointer 1. We obtain in this way $A \sim < B$ in the son string.
- Case (1, 2). The pointer 1 gets *A* and puts it in the son by the left pointer. Later, the pointer 2 meets *A* and skips it, then it gets *B* and transmits it to the son using the left pointer and consequently we obtain $A \sim < B$ in the son string.
- Case (1, 3). The pointer 1 gets *A* and the pointer 3 gets *B*. *A* is posed in the son more to the left respect to *B* because *A* is inserted using the left pointer in the son, *B* using the right pointer and the write pointers cannot cross each other. Then, we get $A \sim < B$ in the son string.
- Case (1, 4). First, the pointer 1 gets *A* and put it in the son by the left write pointer. Then, the pointer 4 gets *B* and put it in the son by the right write pointer. As the write pointers cannot cross each others, then it must be $A \sim < B$ in the son string.

■

The recombination operator proposed is very general indeed. It has four degrees of freedom (the four pointers) we can drive following our wishes. We can combine them in many different configurations so that obtaining recombination operators with very different behaviours. For example, we can think to inhibit a generic combination of two pointers letting free to move only the remaining two. We can think also to bias the random sequences which drive the pointers in order to obtain something more close to the uniform crossover rather than to the one-point crossover or vice versa, biasing in this way the destruction capability of the recombination [7]. Yet we can think to combine two recombination operators presenting complementary aspects during the evolutionary process, applying once the one, once the other, in order to obtain a synergic effect.

In fact we have studied and compared in practice a set of recombination operators selected following the guidelines mentioned above. In this paper we report only one of them, the one which has revealed to be the most effective in our computational experiments. However it is worth mentioning we have found that different recombination operators may affect the success of the GA strongly, especially when the GA is not paired with local search. In our genetic algorithm the Merge and Split recombination (MSX) operator has been used. Figure 5 illustrates by an example how MSX works in practice, its detailed definition follows.

**SEQ: 11212122211221**

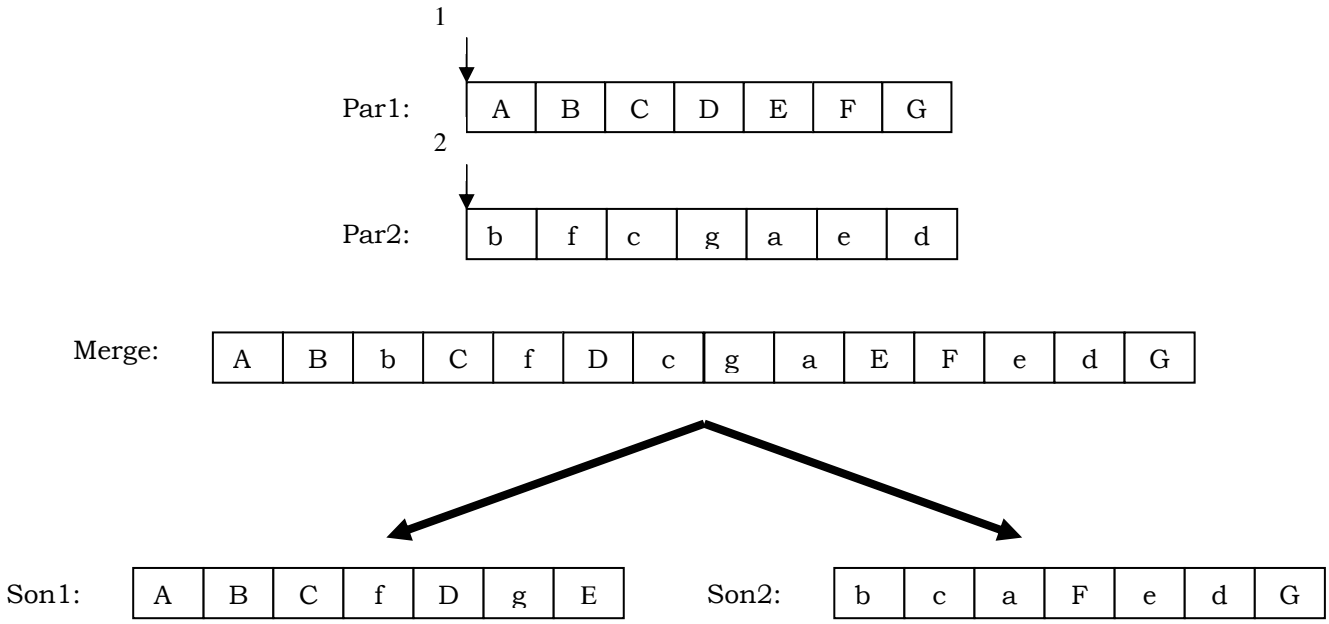


**Figure 5 –** Merge and Split Recombination (MSX)

**Definition 4.** Merge and Split Recombination (MSX).
Let *SEQ* be a vector of $2 \cdot n$ elements randomly chosen in the set {1, 2} such as both elements 1 and 2 occur $n$ times each and where $n$ is the length of strings. We use it as input of the following procedure that produces from two parent strings two offspring strings:

1. Initialise pointers *PT1* and *PT2* at the beginning (on the left side) of parent strings *PAR1* and *PAR2*. Set all operations as unmarked.
2. Consider the first number appearing in the sequence *SEQ*
3. Slide to the right the corresponding pointer to the first unmarked operation
4. Copy the current operation in the *Merge* string in the first position available to the left and mark that operation as already processed
5. Take out the number at the beginning of *SEQ*
6. If *SEQ* is empty then go to step 7 otherwise go to step 2
7. Scan the *Merge* string operation by operation from the left-most to the right-most. The first time an operation is met it is assigned to *Son1*, the second time the same operation is met it is assigned to *Son2* filling them from left to right.

The main peculiarity of MSX consists in getting two complementary sons by combining the precedence characteristics of their parents meanwhile trying to minimise the loss of diversity. More precisely, if the generic operations *a* and *b* have a different order in the parents, such as in parent one *a* precedes *b* and in parent two *b* precedes *a*, MSX tends as much as possible to transmit this diversity to the sons so that in one son *a* will precede *b* and in the other one *b* will precede *a*. It is important to notice that in general this requirement may contrast with the requirement regarding cycling solutions. Therefore, since the string representation doesn't allow to encode cycling solutions, it turns out to be often impossible to get a perfect preservation of parental characteristics through the recombination.

Intuitively, the preservation of diversity through the recombination is roughly explainable by noticing that in the merge phase the precedence characteristics of parents are mixed but not destroyed. Then, in the split phase, the characteristics are repartitioned in two sons and still not destroyed, so that obtaining the original characteristics preserved but combined in a different way.

A pertinent doubt one may have is whether MSX respects the feasibility requirement for recombination stated in Definition 2. After all we have only proven the feasibility for a class of recombination operators (Theorem 4) which seems not to include MSX because of its way of recombining strings in two phases. However we can imagine an alternative definition for MSX such that it results to match the form of the feasible class. The idea is to produce the two twin sons separately, each one in one phase, using the same random sequence twice, once scanning the input sequence and the parent strings from left to right producing one son, once scanning them in the other sense producing the other one.

# 3 Genetic Local Search

*Genetic Local Search Template*

On one hand, problems from combinatorial optimisation are well within the scope of genetic algorithms. Nevertheless, compared to conventional heuristics, genetic algorithms are not well-suited for fine-tuning structures which are very close to optimal solutions. Therefore it is essential to incorporate conventional heuristics into genetic algorithms to construct a more competitive algorithm.

On the other hand, in general the best solution found by a local search algorithm may depend on the initial solution used. However, a multi-start scheme may overcome this problem. As a further refinement, the effectiveness of multi-start iterative approach may

be improved by using the information available from the solutions obtained in the individual cycles. Following this line, several authors have proposed variants of local search algorithms, using ideas from population genetics.
A Genetic Local Search (GLS) algorithm [1] consists of a basic Genetic Algorithm with the addition of a local search optimisation phase applied to every new individual created either in the initial population or during the evolutionary process.
We can give to the GLS algorithm a dual interpretation. On one hand, we can see it as a Genetic Algorithm where Local Search is intended as a smart mutation mechanism. On the other hand, we can see it as structured multi-start mechanism for Local Search where the Genetic Algorithm plays the role of the structure.
However, by seeing the hybrid approach as a whole, Genetic Algorithms are used to perform global exploration among population while Local Search is used to perform local exploitation around chromosomes. Because of the complementary properties of Genetic Algorithms and Local Search which mutually compensate their points of weakness, the hybrid approach often outperforms either method operating alone.
In the following a GLS outline is presented.

**GLS Template**

1. Generate initial population
2. Execute for every individual an initial optimisation by applying local search
3. Assign fitness to every individual
4. Select individuals for recombination
5. Apply the recombination operator producing a new generation of offspring
6. Optimise every new offspring by applying local search
7. Insert the offspring in the population and reduce it to the original size
8. Repeat the steps from 3 to 7 until a stop criterion is met

Let us now fill the Genetic Local Search template presented above with all the components we need to implement an actual algorithm for JSSP. First, we will discuss about the major components of the Genetic Algorithm framework, then we will focus our attention on the specific Local Search algorithm we have used.

*Genetic Algorithm Framework*

- POPULATION. The initial population contains a fixed number of chromosomes which are generated at random. During all the evolutionary process the population size remains constant.

- FITNESS FUNCTION. Every chromosome in the population receives a fitness value. It biases the probability of the chromosome to reproduce. In our case the fitness value of a chromosome is the makespan of its encoded solution.
- SELECTION SCHEME. A fixed number of chromosomes which will undergo recombination are selected. The selection is done via a simple ranking mechanism. The population is always kept sorted according to the fitness. The probability of each chromosome to be selected depends only on its position in the rank and not on the actual fitness value.
- REINSERTION SCHEME. The set of offspring is merged with the population. Then the population is reduced to its original size by eliminating the worst chromosomes.
- STOP CRITERION. The algorithm stops after a fixed numbers of consecutive generations without improvement of the best solution in the population.
- REPRESENTATION & RECOMBINATION. We use the string representation and the MSX recombination operator presented in Section 2. Let us now spend few words on the role played by MSX in the GLS framework, focusing again on its behaviour. While MSX tends to preserve diversity as much as possible, it tries as well to mix parent characteristics a lot. The input sequence is randomly allowed to switch from one parent to the other in every step, therefore it behaves like a uniform crossover. These two aspects of the recombination taken together are particularly welcome in a genetic local search framework. On one hand, MSX transmits the diversity and therefore doesn't trash expensive information present in the parents gathered by local search, the most time-consuming GLS component. On the other hand, the role of the GA paired with local search is to explore as much as possible the solution space. MSX stresses it just shuffling the information present in the parents at most behaving like a uniform crossover.

*Local Search Optimiser*

The TS algorithm here proposed is an effective local search algorithm for JSSP. We use it in the above GLS algorithm as a local search optimisation phase in steps 2 and 6.

**TS Algorithm**

1. Current Solution := Initial solution
2. Best Solution := Initial Solution
3. Taboo List := Empty
4. Consider the neighbourhood of the current solution and select one of them not in the Taboo List following a Search Strategy

5. Insert the current solution in the Taboo List and, if it is full, make room taking out the solution ahead of the list
6. Update the best solution found so far
7. Take the selected neighbour as the new current solution
8. Repeat steps 4-7 until a Stop Criterion is met

More in detail, the Taboo Search we use is based on an algorithm proposed by Eikelder et al [18]. In the following we discuss the major components of the algorithm.

- REPRESENTATION. To apply local search to JSSP we use the disjunctive graph representation. A feasible solution is obtained by orienting the edges such that there is a linear ordering of the operations that have to be processed on one machine, and the resulting graph is acyclic.
- NEIGHBOURHOOD. We use the neighbourhood structure of Nowicki & Smutnicki [13]. It is based on reversing machine arcs on a longest path. However, they have shown that several types of neighbours can be omitted since they cannot have lower costs. For instance it is not useful to reverse internal arcs of a block of operations belonging to a longest path.
- SEARCHING STRATEGY. The time needed to search a neighbourhood depends on the size of the neighbourhood and on the time complexity of the computational cost of neighbours. Since the size of a neighbourhood is rather small we use the steepest neighbour search strategy.
- TABOO LIST. We use a taboo list consisting of a FIFO queue of moves of fixed length. The length of the taboo list is the average neighbourhood size plus a random value.
- STOP CRITERION. The algorithm stops after a fixed numbers of steps without improvement.

Because of the combined use of Genetic Algorithms and Taboo Search we will denote our algorithm with GTS, acronym of Genetic Taboo Search.

# 4 Computational Results

*Relevant Parameters*

In the sequel, we introduce and discuss the most important parameters of GTS, the ones that affect more the performance of the algorithm, and their settings.

COMPUTATIONAL EFFORT

This parameter permits a qualitative control of the computational search effort. More in detail, we define the computational effort as the product of two factors, where the first factor is the number of consecutive iterations without improvement (*TS*) after that each run of Taboo Search has to stop, and the second factor is the number of consecutive individuals processed by the GA without improvement (*GA*) after that GTS has to stop. Since both *TS* and *GA* stop criteria are adaptive to the complexity of the specific problem treated, the setting of the computational effort parameter produces different effects applied on different instances. However, although roughly, it gives a way to control the computational effort.
We have found convenient to set a different computational effort on the basis of the size of the instance treated as shown in Table 2.

*TS/GA* MIXING RATE

This is a very important parameter that is used to weigh the relative contribution of TS and GA. Knowing the Computational Effort (*TS*GA*) and the *TS/GA* ratio we then can determine the stop criteria for TS and GA. We have seen that the bigger the problem is the better GA performs compared with TS. More in detail we have assigned a *TS/GA* ratio of 10:1 for little and medium size instances and 1:1 for large size instances (see Table 2).

| **Size of instances** | ***TS*GA*** | ***TS/GA*** |
|---|---|---|
| Little instances (up to 150 operations) | 10000 | 10:1 |
| Medium instances (around 200 operations) | 100000 | 10:1 |
| Medium-large instances (around 300 operations) | 500000 | 10:1 |
| Large instances (around 500 operations) | 1000000 | 1:1 |

**Table 2 –** Computational Effort and Mixing Rate

GA PARAMETERS

It is very important to set the GA parameters properly in order to guarantee a good flow of information between GA and TS during all the evolutionary process so as to obtain an effective co-operation between them. We have found that the following parameters affect the quality of the flow of information and therefore we have paid great attention in finding a good setting:

- Population Size. We tuned GTS focusing on meaningful relationship among parameters rather than on their absolute values, trying first to find out good ratios among relevant parameters and only later deriving indirectly their absolute values. Following this approach, we have considered the Population Size being in direct relationship with the Number of Generations, obtaining that a good ratio is 1:1. The

absolute values we have found for the Population Size parameter vary gradually from 10 individuals for small size instances up to 50 individuals for large size instances.

- Generational Gap. This parameter represents the number of offspring to produce every generation through recombination. We have found Population Size/2 to be a good setting for this parameter.
- Selection Pressure. This parameter permits to control the competition level in the population. It bias the ranking selection mechanism, making the selection probability of chromosomes more depending or less depending on their rank in the population on the basis of the parameter value. The range of the selection pressure varies from 0 (no dependency) to 2 (strong dependency). A weak selection pressure, therefore, gives a bad individual almost the same chance to reproduce as a good individual, whereas a strong selection pressure strongly favours the reproduction of only good individuals. We have found a weak selection pressure of 0.1 being appropriate for our algorithm. This should not be so surprising because in our GA we use a reinsertion scheme which is already very selective itself, thus making it not necessary to strengthen too much further the selection pressure through this parameter.

*GTS Vs TS*

In Table 3 a direct comparison between the hybrid algorithm GTS and its TS core is presented. This investigation is of crucial importance since we will find out whether the hybridisation is worthy or the genetic framework has just an ornamental function rather than a real merit.

In order to effectuate a fair comparison between GTS and TS we have set parameters in such a way both algorithms get approximately the same amount of time for the same instance. We have applied both algorithms to a set of well-known JSSP instances of various sizes. This set includes the benchmark set introduced by Vaessens [20], which comprises the hard-to-solve Lawrence instances, two of the easier ones and the famous Fisher & Thompson 10×10 instance. Moreover, we test the two algorithms also on three bigger instances, the abz-problems from the test library JSPlib (obtainable via ftp from mscmga.ms.ic.ac.uk). We report the results we have obtained from 10 runs on a Sparc station 5 (110Mhz). The CPU time is expressed in seconds. We can notice that on little instances GTS works as well as TS finding the same quality of solutions and using the same amount of time. As the size of instances increases the GTS works better than TS finding better quality solutions. At first glance TS seems saving time on large instances. This is substantially due to the adaptive stop criteria. In order to overcome this premature termination, we tried to compensate the time difference setting TS in such a way it takes more time, therefore giving it the chance of getting better solutions. TS gets stuck anyway without improving the solution quality, thus wasting all the additional time we gave to it.

*A Wide Comparison*

We have done a wide comparison on well-known instances among GTS and the best algorithms belonging to a variety of approaches proposed by Vaessens [20]. Table 4 gives the best costs found by GTS and other methods. In general we see that GTS behaves very well. Again we see that with big instances GTS outperforms all the other approaches.
In the following we list the programs we have considered:

- RGLS-5 – Reiterated Guided Local Search by Balas and Vazacopoulos [2]
- TS-B – Taboo Search and Backtracking by Nowicki and Smutnicki [13]
- SA1 – Simulated Annealing by Van Laarhoven, Lenstra and Ulder [1]
- SB-GLS – Shifting Bottleneck and Guided Local Search by Balas and Vazacopoulos [2]
- GA-SB – Genetic Algorithms and Shifting Bottleneck by Dondorf and Pesch [5]

*GTS Vs SAGen*

Finally we have done a specific comparison between our hybrid algorithm (Taboo Search Based) and another recent hybrid genetic algorithm based on Simulated Annealing proposed by Kolonko [8].
As we can see in Table 5, we have compared these two algorithms on the set of difficult swv instances from JSPlib, almost all still open, setting the stop criteria preferring quality against time. We report the results of 3 runs for both algorithms, on a Sparc station 5 (110Mhz) for GTS and on Pentium 120/166Mhz for SAGen. The time is expressed in seconds. As we can see both on the quality and time GTS strongly outperforms SAGen and most of the times GTS breaks the known bound for those instances.

| Problem | OPT (UB) | BEST GTS | AVG GTS | BEST TS | AVG TS | AVG TIME GTS | AVG TIME TS |
|---|---|---|---|---|---|---|---|
| **10 jobs * 5 machines = 50 operations** | | | | | | | |
| la02 | 655 | 655 | 655 | 655 | 663 | 5 | 9 |
| **10 jobs * 10 machines = 100 operations** | | | | | | | |
| ft10 | 930 | 930 | 933 | 930 | 933 | 67 | 65 |
| la19 | 842 | 842 | 842 | 842 | 842 | 41 | 47 |
| **15 jobs * 10 machines = 150 operations** | | | | | | | |
| la21 | 1046 | 1047 | 1050 | 1048 | 1063 | 117 | 122 |
| la24 | 935 | 938 | 943 | 942 | 943 | 85 | 140 |
| la25 | 977 | 977 | 978 | 977 | 978 | 91 | 190 |
| **20 jobs * 10 machines = 200 operations** | | | | | | | |
| la27 | 1235 | 1235 | 1240 | 1255 | 1264 | 257 | 140 |
| la29 | (1153) | 1157 | 1166 | 1167 | 1177 | 316 | 233 |
| **15 jobs * 15 machines = 225 operations** | | | | | | | |
| la36 | 1268 | 1268 | 1274 | 1268 | 1276 | 184 | 197 |
| la37 | 1397 | 1403 | 1410 | 1415 | 1420 | 229 | 208 |
| la38 | 1196 | 1201 | 1202 | 1199 | 1204 | 178 | 275 |
| la39 | 1233 | 1233 | 1239 | 1233 | 1247 | 207 | 220 |
| la40 | 1222 | 1226 | 1231 | 1229 | 1232 | 192 | 211 |
| **20 jobs * 15 machines = 300 operations** | | | | | | | |
| abz07 | (656) | 658 | 662 | 666 | 668 | 1764 | 975 |
| abz08 | (669) | 670 | 672 | 680 | 681 | 1518 | 931 |
| abz09 | (679) | 682 | 687 | 688 | 689 | 1250 | 1114 |

**Table 3 –** GTS Vs TS

| Problem | OPT | GTS | RGLS-5 | TS-B | SA1 | SB-GLS | GA-SB |
|---|---|---|---|---|---|---|---|
| la02 | 655 | 655 | 655 | 655 | - | 666 | - |
| ft10 | 930 | 930 | 930 | 930 | - | 930 | - |
| la19 | 842 | 842 | 842 | 842 | - | 852 | 848 |
| la21 | 1046 | 1047 | 1046 | 1047 | 1053 | 1048 | 1074 |
| la24 | 935 | 938 | 935 | 939 | 935 | 941 | 957 |
| la25 | 977 | 977 | 977 | 977 | 983 | 993 | 1007 |
| la36 | 1268 | 1268 | 1268 | 1268 | - | 1268 | 1317 |
| la37 | 1397 | 1403 | 1397 | 1407 | - | 1397 | 1446 |
| la38 | 1196 | 1201 | 1196 | 1196 | 1208 | 1208 | 1241 |
| la39 | 1233 | 1233 | 1233 | 1233 | - | 1249 | 1277 |
| la40 | 1222 | 1226 | 1224 | 1229 | 1225 | 1242 | 1252 |
| la27 | 1235 | 1235 | 1235 | 1236 | 1249 | 1243 | 1269 |
| la29 | 1142 / 1153 | 1157 | 1164 | 1160 | 1185 | 1182 | 1210 |

**Table 4 –** Wide Comparison

| Problem | OPT LB / UB | BEST GTS | AVG GTS | BEST SAGen | AVG SAGen | AVG TIME GTS | AVG TIME SAGen |
|---|---|---|---|---|---|---|---|
| **20 jobs * 10 machines = 200 operations** | | | | | | | |
| swv01 | 1392 / 1418 | 1430 | 1430 | 1427 | 1428 | 2034 | 47828 |
| swv02 | 1475 / 1491 | 1481 | 1484 | 1487 | 1490 | 1851 | 43089 |
| swv03 | 1369 / 1398 | 1418 | 1425 | 1422 | 1428 | 2102 | 40684 |
| swv04 | 1450 / 1493 | 1482 | 1488 | 1487 | 1490 | 2315 | 44257 |
| swv05 | 1421 / 1448 | 1441 | 1447 | 1449 | 1453 | 1924 | 40045 |
| **20 jobs * 15 machines = 300 operations** | | | | | | | |
| swv06 | 1591 / 1718 | 1701 | 1710 | 1697 | 1703 | 3536 | 112647 |
| swv07 | 1446 / 1652 | 1625 | 1626 | 1627 | 1630 | 4394 | 97504 |
| swv08 | 1640 / 1798 | 1774 | 1781 | 1773 | 1776 | 4105 | 56781 |
| swv09 | 1604 / 1710 | 1675 | 1686 | 1665 | 1682 | 3667 | 24474 |
| swv10 | 1631 / 1794 | 1775 | 1780 | 1791 | 1794 | 5556 | 44467 |
| **50 jobs * 10 machines = 500 operations** | | | | | | | |
| swv11 | 2983 / 3047 | 3019 | 3025 | 3075 | 3081 | 13262 | 117454 |
| swv12 | 2972 / 3045 | 3040 | 3071 | 3108 | 3115 | 16029 | 124549 |
| swv13 | 3104 / 3173 | 3107 | 3116 | 3177 | 3178 | 14420 | 92756 |
| swv14 | 2968 | 2968 | 2971 | 3010 | 3013 | 10951 | 104088 |
| swv15 | 2885 / 3022 | 2918 | 2929 | 3004 | 3004 | 16773 | 161365 |

**Table 5 –** GTS Vs SAGen

# 5 Conclusions

This paper describes an hybrid algorithm (GTS) combining Genetic Algorithms and Taboo Search for the JSSP. The ingredients of our GA are a natural representation of solutions (the string representation) and a recombination capable of transmitting meaningful characteristics (the common order relationship) from parents to children. The problems of feasibility regarding cycling and job constraints have been discussed and solved in that framework. Moreover, the MSX recombination operator that tends to preserve the diversity of the parent schedules in the offspring schedules has been presented. The Genetic Local Search scheme has been used to hybridise our GA with an effective TS algorithm for JSSP. Computational experiments have shown that on large size instances the GA counterpart makes indeed the difference. The best mix of TS and GA for those instances is half and half (following our mix definition) and therefore GTS has to be considered as a real hybrid, neither a modified TS nor a modified GA. GTS has been compared with a variety of other approaches and it has revealed to perform very well in the comparison. The last experiment has shown that GAs are far more profitably

hybridised with Taboo Search than with Simulated Annealing. As a matter of fact both on time required and solution quality a difference of one order of magnitude has been found.

Let us spend some more words on the philosophy underlying the natural approach we use. The crucial point is that we see a schedule as a partial order relationship among operations. It is not important that the relationship is made of contributes from precedence constraints given with the problem instance and those ones given with the particular solution to that problem. We see all the constraints uniformly without any distinction, all forming the relationship among operations.
By seeing schedules like relationships, it is natural to think about recombination as a way to recombine partial order relationships transmitting to son schedules the common sub-relationship of parent schedules. This seems a natural requirement as we are considering schedules at this level. As a welcome side-effect of this approach, we obtain that in the transmission of meaningful characteristics to sons even the feasibility property (intended as the job precedence constraints of the problem instance) is transmitted from parents to sons without paying special attention to it. We treat it uniformly as a generic characteristic of a schedule. This positive side-effect leaves us thinking we are approaching the problem at the right level of abstraction without being misled by the syntactical details of the representation used. Finally a further consequence of the way we approach the problem is that the string representation and the recombination proposed do not depend on a particular configuration of the constraints and therefore they can be naturally extended to more general scheduling problems.

# References

[1] E. H. L. Aarts, P. J. M. van Laarhoven, J. K. Lenstra, N. L. J. Ulder – A computational study of local search algorithms for job shop scheduling – In: *ORSA Journal on Computing* Vol. 6, No. 2, Spring 1994.
[2] E. Balas and A. Vazacopoulos – Guided Local Search with Shifting Bottleneck for Job Shop Scheduling – In: *Management Science Research Report* #MSRR-609, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania.
[3] L.D. Davis – *Handbook of Genetic Algorithms* – Van Nostrand Reinhold, 1991.
[4] F. Della Croce, R. Tadei, G. Volta – A Genetic Algorithm for the Job Shop Problem – In: *Computers and Operations Research* Vol. 22, No. 1, 1995, pp. 15-24.
[5] U. Dorndorf and E. Pesch – Evolution Based Learning in a Job Shop Scheduling Environment – *Computer and Operations Research* 22, 1995, pp. 25-40.

[6] F. Glover, C. McMilan and B. Novick – Tabu Search, Part I – *ORSA J. Computing*, 1, 3, 1989, pp. 190-206.
[7] D.E. Goldberg – *Genetic Algorithms in Search, Optimisation and Machine Learning* – Addison Wesley Publishing Company, January 1989.
[8] M. Kolonko - Some new results on simulated annealing applied to the job shop scheduling problem – In: *European Journal of Operational Research* Vol. 113, No. 1, 1999, pp. 123-136.
[9] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys. – Sequencing and scheduling: Algorithms and complexity. – In: S.C. Graves, A.H.G. Rinnoy Kan and P. Zipkin, editors, *Handbooks in Operations Research and Management Science* 4, North-Holland, 1993.
[10] Z. Michalewicz – *Genetic Algorithms + Data Structures = Evolution Programs* – Springer-Verlag, AI Series, New York, 1996.
[11] H. Mühlenbein, M. Gorges-Schleuter and O. Krämer – Evolution Algorithms in Combinatorial Optimisation – In: *Parallel Computing* 7, 1988, pp. 65-85.
[12] R. Nakano, T. Yamada – Conventional genetic algorithm for job shop problems – *Proceedings of 4$^{th}$ ICGA*, 1991, pp. 474-479.
[13] E. Nowicky and C. Smutnicki - A fast taboo search algorithm for the job shop problem – In: *Management Science* Vol. 42, 6, June 1996.
[14] J.T. Richardson, M.R. Palmer, G.E. Liepins, M.R. Hilliard – Some guidelines for genetic algorithms with penalty functions – In: J.D. Shaffer editor, *Proceedings of the third international conference on genetic algorithms*, Morgan Kaufmann, 1989, pp. 191-197.
[15] B. Roy, B. Sussmann – Les problèmes d'ordonnancement avec contraints disjonctives – Note DS 9 bis, SEMA, 1964, Paris, France.
[16] E. Taillard – Parallel taboo search techniques for the job shop scheduling problem – *ORSA J. Computation*, 6, 1994, pp. 108-177.
[17] A.Y.C. Tang and K.S. Leung – A Modified Edge Recombination Operator for the Travelling Salesman Problem – In: H.-P. Schwefel and Manner editors, *Parallel Problem Solving from Nature III*, Springer-Verlag, 1994, pp. 180-188.
[18] H.M.M. ten Eikelder, B.J.M. Aarts, M.G.A. Verhoeven, E.H.L. Aarts – Sequential and Parallel Local Search Algorithms for Job Shop Scheduling – In: S. Voss, S. Martello, I.H. Osman and C. Roucairol, editors: *Meta-Heuristics, Advances and Trends in Local Search Paradigms for Optimization*, Kluwer, 1999, pp. 359-371.
[19] N.L.J. Ulder, E.H.L. Aarts, H-J. Bandelt, P.J.M. van Laarhoven and E. Pesch – Genetic Local Search for the Travelling Salesman Problem – In: *Lecture Notes in Computer Science* 496, Springler, 1990, Berlin, pp. 109-116.

[20] R.J.M. Vaessens, E.H.L. Aarts and J.K. Lenstra - Job shop scheduling by local search – In: *INFORMS Journal on Computing* Vol. 8, No. 3, Summer 1996, pp. 302-317.