1-1-2015

An effective asexual genetic algorithm for solving the job shop scheduling problem

Mehrdad Amirghasemi
University of Wollongong, mehrdad@uow.edu.au

Reza R. Zamani
University of Wollongong, reza@uow.edu.au

An effective asexual genetic algorithm for solving the job shop scheduling problem

Abstract

All rights reserved. Abstract By using the notion of elite pool, this paper presents an effective asexual genetic algorithm for solving the job shop scheduling problem. Based on mutation operations, the algorithm selectively picks the solution with the highest quality from the pool and after its modification, it can replace the solution with the lowest quality with such a modified solution. The elite pool is initially filled with a number of non-delay schedules, and then, in each iteration, the best solution of the elite pool is removed and mutated in a biased fashion through running a limited tabu search procedure. A decision strategy which balances exploitation versus exploration determines (i) whether any intermediate solution along the run of tabu search should join the elite pool, and (ii) whether upon joining a new solution to the pool, the worst solution should leave the pool. The genetic algorithm procedure is repeated until either a time limit is reached or the elite pool becomes empty. The results of extensive computational experiments on the benchmark instances indicate that the success of the procedure significantly depends on the employed mechanism of updating the elite pool. In these experiments, the optimal value of the well-known 10 x 10 instance, ft10, is obtained in 0.06 s. Moreover, for larger problems, solutions with the precision of less than one percent from the best known solutions are achieved within several seconds.

Keywords

problem, effective, asexual, genetic, algorithm, solving, job, shop, scheduling

Disciplines

Engineering | Science and Technology Studies

Publication Details

Amirghasemi, M. & Zamani, R. R. (2015). An effective asexual genetic algorithm for solving the job shop scheduling problem. Computers and Industrial Engineering, 83 123-138. Computers and Industrial Engineering

Mehrdad Amirghasemi & Reza Zamani

An effective asexual genetic algorithm for solving the job shop scheduling problem

Abstract

By using the notion of elite pool, this paper presents an effective asexual genetic algorithm for solving the job shop scheduling problem. Based on mutation operations, the algorithm selectively picks the solution with the highest quality from the pool and after its modification, it can replace the solution with the lowest quality with such a modified solution. The elite pool is initially filled with a number of non-delay schedules, and then, in each iteration, the best solution of the elite pool is removed and mutated in a biased fashion through running a limited tabu search procedure. A decision strategy which balances exploitation versus exploration determines (i) whether any intermediate solution along the run of tabu search should join the elite pool, and (ii) whether upon joining a new solution to the pool, the worst solution should leave the pool. The genetic algorithm procedure is repeated until either a time limit is reached or the elite pool becomes empty. The results of extensive computational experiments on the benchmark instances indicate that the success of the procedure significantly depends on the employed mechanism of updating the elite pool. In these experiments, the optimal value of the well-known 10x10 instance, ft10, is obtained in 0.06 seconds. Moreover, for larger problems, solutions with the precision of less than one percent from the best known solutions are achieved within several seconds.

1. Introduction

Evolutionary search techniques, in general, and genetic algorithms, in particular, can be considered as machine-learning techniques aiming at effective process of strings for finding

beneficial patterns which contribute to solution quality. The rationale behind genetic algorithms is the notion of schemata, which can be considered as similar patterns existing in the encoding of a number of solutions.

Considering a genome as a point in a multi-dimensional space, schemata are similar to hyper-planes, covering a specific pattern of points. By the means of crossover operators, genetic algorithms typically search for advantageous schemata which contribute to solution quality. In such a typical setting, mutation operators, despite their importance, are usually used in the background solely to diversify the pool of solutions.

In this paper, a genetic algorithm has been presented which tackles the job shop scheduling problem by using mutation operators in the foreground of the search. In other words, the algorithm, which avoids using any crossover operator, employs the "survival of the fittest" principle in an asexual reproduction scheme.

The development of this genetic algorithm has been based on three conjectures. First, the success of any metaheuristic search procedure, from point-based to population-based, depends on the trade-off it makes between exploration and exploitation, with exploration aiming at exploring new regions, and exploitation aiming at searching the high-quality regions already distinguished. Second, in genetic algorithms, it is the combination of mutation operators and the mechanisms manipulating the population which highly affects the trade-off between exploration and exploitation. Third, biased-mutations can have a twofold role in the sense of contributing to both exploration and exploitation. In other words, through a biased-mutation not only can the search be diversified, but the search can be guided towards high quality solutions as well.

Based on these conjectures, in this paper, the lack of a crossover operator has been compensated by an effective manipulation of the population and a biased-mutation which favors both

exploration and exploitation. In other words, instead of requiring any crossover operator to tunnel the routes which the mutation can traverse directly, the route becomes straightforward towards the goal by a biased-mutation.

The corresponding mutation is extensive and is performed through running a limited tabu search on initial solutions which have been selected among elite solutions. Hence, if before this mutation any crossover operator is performed, it degrades the quality of the initial solutions and consequently leads to overall solutions with lower quality. In other words, in this context, crossover operators destroy the effort expended in creating elite solutions.

Since the main handicap of all genetic algorithms is the incapability of the fine-tuning of solutions, the incorporation of a local search in the presented genetic algorithm has dual benefits. On the one hand, it fine-tunes the solutions, and on the other hand, it can be employed in generating the biased-mutation, which can favor exploitation along with exploration. For this reason, a tabu search has been integrated in the employed asexual genetic algorithm, and the consequent procedure, called TGA (Tabu-based Genetic Algorithm), improves the quality of a pool of solutions, called elite pool, by biased-mutation operations proposed by its tabu search component.

At each stage, the solution with the highest quality in the elite pool is fine-tuned by the tabu search component and, in this process, intermediate solutions generated either can potentially replace the solution with the lowest quality in the pool or can be added to the pool without omitting any existing solution. The TGA first fills the pool with a number of non-delay schedules generated by the Giffler and Thompson method.

Determining whether or not an intermediate solution along the run of tabu search should join the elite pool is made by a decision strategy which also determines whether the worst solution

should remain in the pool. The TGA is terminated whenever either a time limit has been reached or the elite pool has become empty. The mechanism of updating the elite pool is aimed at striking a balance between the factors of exploration and exploitation, and the biased-mutation mechanism employed contributes to increasing both of these critical factors simultaneously.

The rest of the paper is as follows. Section 2 presents a formulation for the job shop scheduling problem, and Section 3 describes the related work. Section 4 presents the TGA and provides a stepwise description of its algorithm. The results of computational experiments are discussed in Section 5. The concluding remarks are discussed in Section 6, which also sketches several possible directions for future work.

2. Problem Formulation

The job shop scheduling problem (JSP) is defined as a collection of n jobs which should be processed on a set of m machines, with each job having a predetermined order on different machines. The goal in this problem is to minimize the makespan, which is defined as the completion time of the last job completed. For the purpose of simplicity, each job j is usually considered as a series of operations, $O_{j1,}O_{j2,...,}O_{jm}$, which should be completed one after another, each on a different machine.

The required machine and execution time for the operation O_{jk} are denoted by Ω_{jk}, and τ_{jk}, respectively. For instance, when Ω_{36} is equal to 8 and τ_{36} is equal to 5, the 6^{th} operation of the job 3 requires machine 8 and takes 5 units of time. Moreover, once started, an operation cannot be interrupted and should continue unti l it has been completed.

Hence, two values m and n and two matrices Ω and τ, each with the dimension of $n\times m$,are the entire requirements needed to describe the problem. The final output of any procedure solving

the JSP is a single $n \times m$ matrix in which the starting time of each operation O_{jk} has been identified as S_{jk}. The JSP can be formulated as:

$$\min_{for\ all\ feasble\ starting\ times} \max(S_{jk} + \tau_{jk} : 1 \le j \le n\ \&\ 1 \le k \le m) \quad (1)$$

Subject to:

$$S_{jk} + \tau_{jk} \le S_{j\,(k+1)} \quad : (1 \le j \le n\ \&\ 1 \le k \le m-1) \quad (2)$$

$$S_{jk'} + \tau_{jk'} \le S_{j'k}\ or\ S_{j'k} + \tau_{j'k} \le S_{jk'} \quad : (\Omega_{jk'} = \Omega_{j'k}\ \&\ 1 \le j\ \&\ j' \le n\ \&\ 1 \le k\ \&\ k' \le m) \quad (3)$$

Whereas the first constraint implies that, based on the given order, the operations of each job should be executed one after another, the second constraint prevents any machine from simultaneously performing more than one operation. The above formulation, which is a type of disjunctive graph formulation, views the JSP as finding the order of operations which require the same machine. In effect, the disjunctive graph formulation is one of the most effective formulations of the JSP and, since its early development, has predominately used by the researchers to tackle the JSP.

In a disjunctive graph, nodes represent operations and arcs represent the precedence relations. The cost of each arc is equal to the duration of its starting operation. Arcs are divided into two groups of the conjunctive and disjunctive. Whereas conjunctive arcs are fixed based on the first constraint and their fixation is performed in the initialization of the graph, the fixing of the disjunctive arcs determines the order of different operations on each specific machine, and hence, represent the solution of the problem. After the fixation of the disjunctive arcs, the length of longest path in the disjunctive graph shows the makespan of the problem and the longest path from the starting node of the graph to each operation shows the starting time of the

corresponding operation. For the purpose of simplicity, in the formulation presented, we use the notation of φ_i^j to denote an operation of the job *j* which requires machine *i*.

$$O_{jk} = \varphi_i^j \leftrightarrow \Omega_{jk} = i \tag{4}$$

The predecessor and successor of operations φ^j are denoted with $jp(\varphi_i^j)$, and $js(\varphi_i^j)$, respectively. In the cases where an operation is the starting or ending operation of its job, its *jp* or *js* is considered as null, respectively. The following equations show how *js* and *jp* are defined.

$$js\left(\varphi_i^j\right) = \varphi_l^j \leftrightarrow \exists\delta \mid \Omega_{j\delta} = i \ \Lambda \ \Omega_{j\delta+1} = l \tag{5}$$

$$jp\left(\varphi_i^j\right) = \varphi_l^j \leftrightarrow \exists\delta \mid \Omega_{j\delta} = i \ \Lambda \ \Omega_{j\delta-1} = l \tag{6}$$

Figure 1 shows a sample JSP problem and its disjunctive graph. Two operations φ_{i1}^{j1} and $\varphi_{i\,2}^{j2}$ which have no precedence relation in the disjunctive graph can be executed in the overlapping intervals. The case where φ_{i1}^{j1} should be completed before the start of φ_{i2}^{j2} is represented with $\varphi_{i1}^{j1} \prec \varphi_{i\,2}^{j2}$. Whenever $j_1 \neq j_2$ and $i_1 \neq i_2$ there is no need to set any precedence relation between φ_{i1}^{j1} and φ_{i2}^{j2}, because different machines can simultaneously work on different operations of the different jobs. Moreover, when j_1=j_2 the input matrix Ω has initially set precedence relation between φ_{i1}^{j1} and φ_{i2}^{j2}.

Figure 1 is inserted here

Hence, the only precedence relations that are required to be set by any JSP procedure is for the case i_1=i_2. In other words, all operations performed by the same machine need a proper order. For machine *i*, we represent this order as π_i. For instance, π_2={3,4,2,1} implies that on machine

2, jobs 3, 4, 2, and 1 should be executed one after another. Hence solving the JSP is equivalent of finding a feasible $\Pi = \{\pi_1, \pi_2,....,\pi_m\}$ which can minimize the value of the makespan. The word feasible has been used to indicate that when Π and Ω are considered together, no contradiction can be inferred. For example, the following precedence relations indicate a contradiction:

$$\Omega: \varphi_2^1 \prec \varphi_1^1 \ \& \ \varphi_1^2 \prec \varphi_2^2 \quad \Pi: \varphi_2^2 \prec \varphi_2^1 \ \& \ \varphi_1^1 \prec \varphi_1^2 \tag{7}$$

In other words, based on the terminology used in the disjunctive graph, Π should not introduce any loop to the graph. It is worth noting that this limitation does not make the problem harder but easer. After all, it limits Π and makes many of solutions infeasible, reducing the size of solution space significantly. Figure 2 shows the occurrence of a loop in fixing the disjunctive arcs of the sample JSP.

Figure 2 is inserted here

Now assume that a feasible $\Pi = \{\pi_1, \pi_2,....,\pi_m\}$ has been given, and the disjunctive arcs of the graph have been fixed based on Π. The fixation of disjunctive arcs determines the starting time of activities as well as the makespan of the problem, which is not necessarily optimal. The goal of any local search is to improve Π repeatedly so that the corresponding makespan become smaller and possibly optimal. Figure 3 shows, one of many different possible ways in which the disjunctive arcs of the sample project can be fixed. As is seen, in this fixation, the makespan of 20 has been obtained.

Figure 3 is inserted here

Note that by imposing Π, every operation φ_i^j can have a machine successor and machine predecessor operation as well. These two operations are shown with $mp(\varphi_i^j)$ and $ms(\varphi_i^j)$, respectively, representing the immediate operation that precedes (succeeds) the operation φ_i^j on

machine *i*. In the cases where an operation is the starting or ending operation of its machine, its *mp* or *ms* is null, respectively.

Hence, each operation φ_i^j has between zero and two predecessors, $jp(\varphi_i^j)$ & $mp(\varphi_i^j)$, and between zero and two successors, $js(\varphi_i^j)$ & $ms(\varphi_i^j)$. We also define the head of each operation, φ_i^j, as the longest path from the starting node of the disjunctive graph to it, $head(\varphi_i^j)$, and define its tail as the longest path from it to the ending node of the disjunctive graph, $tail(\varphi_i^j)$. The processing time of φ_i^j is denoted with $t(\varphi_i^j)$ and the longest path in the disjunctive graph induced with Π is shown with *makespan* (Π).

3. Related work

In the absence of effective exact solution procedures for the JSP, heuristic methods dominate the literature of this problem. With respect to heuristics, the three categories of construction methods, point-based, and pool-based solution strategies are considered as three distinct categories in solving combinatorial optimization problems, in general, and the JSP, in particular. Construction methods build a solution in an incremental manner, point-based methods improve a single complete solution, and pool-based methods use a pool of solutions and use crossover as well as mutation operators to change the pool from one generation to another.

Construction methods can produce initial solutions, pool-based methods combine the initial solutions in the hope of obtaining better solutions, and point-based methods fine-tune a solution. In effect, the lack of fine-tuning capability in the pool-based strategies can be compensated through the fine-turning power of point-based strategies, with the construction methods producing the initial solutions needed for this amalgamation. Hence, it is mainly the combination of these three approaches that can produce high quality solutions; in this section, we briefly review related work in the three categories of construction, point-based, and pool-based methods.

Perhaps the most important construction method for the JSP is the Shifting Bottleneck Procedure (SBP) presented in (Adams et al., 1988). This procedure sequences the machines by the one-machine scheduling method developed in (Carlier, 1982), which finds the optimal schedule for the corresponding machine which is the relaxation of the JSP. This not only is used for sequencing each single machine but it is used for consecutively ranking the machines as well.

After sequencing the machine with the highest rank, the procedure sequences all of the sequenced machines again. In other words, the SBP repeatedly relaxes the problem and solves the relaxed problem to optimality until it finds a high quality solution for the entire problem. In ranking the remaining machines for selecting the one with the highest rank for sequencing, the SBP solves each of the remaining machines independently, with the Carlier's method, and selects the machine which has led to the highest value of the objective function. Such a critical machine is referred to as bottleneck machine.

In effect, sequencing machines based on their criticality has an essential role in increasing the quality of the overall solution produced by the SBP. The reason for giving priority to sequencing these bottleneck machines in early stages is that their sequencing in later stages can incur higher increase in the overall cost. Since there are other perspectives to view the criticality of machines, and each other proper ranking of machines can lead to solutions with different quality, the SBP can be employed in different frameworks. In this way, the SBP can expand the search tree to possibly find the optimal solution of the problem, albeit at the expense of high computational time and without any guarantee of finding a solution.

It is in this framework that, after 25 years of extensive research by different researchers for finding the optimal solution of ft10, such a solution was obtained. The optimality was guaranteed because of the fact that the lower bound provided by the bottleneck machine in the first level of

the search tree was equal to the upper bound calculated in the search process. This indicates how a one machine problem can be a significantly tight relaxation of the JSP. Part of the importance of this method in the literature is due to this significant tightness.

The second construction method discussed is the method presented in (Giffler & Thompson, 1960). The authors first have shown that for the JSP, the optimal solution is amongst active schedules. Then, they have presented an algorithm that can generate such schedules. With a simple modification, the algorithm is also capable of generating non-delay schedules, which on average have higher quality than active schedules but their set may not include any optimal solution.

In the Giffler and Thompson's algorithm (GT), first, for each machine, the set of eligible operations and their potential start times are determined. Then among all these operations, the one which can be completed in the earliest possible time, t, is specified and its corresponding machine will be selected as a machine for which the algorithm will find the next operation. This operation is simply chosen based on the given priority among all of its eligible operations that could be started before t. The operation is scheduled and its successor joins to the eligible operations of the machine this successor requires.

The potential start times of the machine on which the operation was completed are also updated. The process continues until all operations are scheduled. The generated schedule is guaranteed to be an active schedule. In order to produce a non-delay schedule, a simple modification is performed as follows. Among all of the eligible operations, an operation is selected whose starting time is not greater than $t-d$, with d showing the duration of the longest operation that can be completed at t.

The third related work in the construction methods is the bi-directional insertion method presented in (Dell'Amico & Trubian, 1993). The method includes both forward and backward insertion. In forward insertion, operations are built into the schedule one after another and fix the disjunctive arcs of the disjunction graph. The order in which operations are built into the schedule is based on the so called least-worsening rule.

Based on this rule, first all disjunctive arcs are removed from the disjunctive graph and the length of the corresponding longest path is calculated. Then progressively, among the eligible operations, the disjunctive arc of an operation is added to the disjunctive graph. Without creating any loop, the selected disjunctive arc incurs minimum increase in the length of the critical path. This continues until the order of all operations on all machines is determined.

Forward insertion encounters the lack of proper operations in its later stages of construction. In other words, whereas in its earlier stages it can find arcs that incur very small increases in the longest path, the arcs selected in the later stages are involved with significant increases. That makes the implementation of backward insertion necessary, in which simply arcs are added in the corresponding mirror disjunctive graph. With running forward and backward insertion one after another, the drawback mentioned is circumvented and the method can produce proper results. Several Insertion methods have been presented in (Werner & Winkler, 1995) and compared with one another.

Having discussed three related construction methods, we now describe the second category, which includes point-based strategies. The efficiency of the point-based strategies for each combinatorial optimization problem depends of the landscape of the corresponding problem. In (Bierwirth et al., 2004) a novel analysis of the fitness landscape of the JSP has been performed, and it has been shown that the landscape of the JSP, unlike those of many other combinatorial

optimization problems is non-regular. Here by non-regularity, the authors mean a bias in connectivity. This means that such a bias can influence random walks in general and local searches, in particular.

The prerequisite of measuring auto-correlation for finding the degree of ruggedness of a landscape is that all elements of the landscape can be visited through random walk by equal probability. It should be noted that a local search probes the search space through the signals it receives from the fitness landscape. Since the areas with high degree of connectivity show a better mean fitness, based on having a non-regular landscape, even random walk should enhance solution quality to some extent.

A simulated annealing has been presented in (Van Laarhoven et al., 1992) which has been proven to asymptotically converge to a globally minimum solution. The famous neighborhood, N1, has been presented in that paper. With respect to N1, it has been shown that it never leads to a cyclic disjunctive graph. Two other important neighborhood structures are N5 and N6. The N5 neighborhood (Nowicki & Smutnicki, 1996) interchanges the first two or the last two operations of each block with the exceptions of the first and last blocks. In effect, in the first and the last blocks only the last and first two operations are swapped, respectively. On the other hand, the N6 neighborhood (Balas & Vazacopoulos, 1998) moves each operation of a block after the last or before the first operation of the block, if feasible.

In (C.Y. Zhang et al., 2007) a tabu search with a new neighborhood structure has been presented that works with an efficient move evaluation strategy to tackle the JSP. They have shown that initial solutions have insignificant effect on the performance of their algorithms and that is why they have used randomly generated solutions. The neighborhood structure they have used, N6′, is an extension of N6, in the sense that it allows N6 to move the first or the last operation of the

block into the interior operation inside the block. In the employed tabu list, both the sequence of operations and their positions on the machines are memorized.

One of the most effective point-based procedures for the JSP has been presented in (Nowicki & Smutnicki, 1996); this procedure is based on a Tabu search which uses a substantially small neighborhood. The corresponding neighborhood structure, which limits the moves in N1 to those involved with one of the ending block operations, removes many of unfruitful moves of N1, and keeps only those moves which have a chance of improving the current solution.

This effective neighborhood not only limits the search to promising parts of the search space but makes the evaluation of each move fast as well. In (Jain et al., 2000), this efficient algorithm has been fully examined and seven components contributing to its effectiveness have been identified. These components include (i) initial solution, (ii) move selection, (iii) tabu list, (iv) elite-solutions maintenance, (v) cycle check, (vi) elite solution recovery, and (vii) parameter selection. After examining these seven components, the authors have concluded that the effectiveness of the procedure is mainly due to (i) the move selection component and the corresponding restricted neighborhood structure, (ii) the initial solution component which can create semi-active solutions, and (iii) the saving as well as the recovery of elite solutions. They also showed that, in the process of the corresponding Tabu search, over 99.7 percent of moves generated with this neighborhood structure are disapproving.

Moreover, despite the well-known fact that active schedules on average have better quality than semi-active schedules, this study shows that one of the contributors to the efficiency of this algorithm is the use of semi-active schedules. This counter intuitive result can be partly described by the fact that semi active schedules have a greater number of neighbors than active schedules. In effect, since the employed neighborhood structure highly restricts the size of the

neighbors, it is the combination of this neighborhood structure and comparatively large size of neighbors which makes their combination effective.

Their study also shows that despite using a very effective neighborhood structure, a majority of the search effort is spent on evaluating moves which are disapproving. Hence accelerating the evaluation stage, at the cost of slight imprecision of evaluation result, may circumvent the dilemma mentioned. However, the development of an approximate evaluation technique that is both comparatively fast and significantly accurate is a complicated task. The methods presented in (Taillard, 1994), (Dell'Amico & Trubian, 1993), and (Nowicki & Smutnicki, 2005) are examples of such estimation strategies.

The big valley property of landscape has been exploited in (Nowicki & Smutnicki, 2005) to develop an effective tabu search procedure that uses some elements of path relinking technique in tackling the JSP. Towards reducing the computational cost of exact estimation of moves of N1 and faster elimination of critical paths, the procedure uses an effective technique based the following two cases. Assuming that on the critical path the move has caused the disjunctive arc connecting operation α to β to reverse, either the new critical path includes α or not. In each of these two cases, a fast and simple exact estimation procedure has been provided.

In the path relinking part, two elite processing orders are considered as the extremes of a spectrum and the best processing order between these two extremes is found. For this purpose, all processing orders in this spectrum are required to be computed. The number of swaps needed to convert one processing order to the other shows the distance between the two processing orders, and the larger the distance between these two extremes, the larger the number of possible processing orders in the spectrum.

With respect to repetitive constructing of the disjunctive graph, one of the main points that contribute to the effectiveness of this algorithm is the mechanism by which the topological orders as well as the heads and tails of operations are calculated. This mechanism works based on the fact that if the locations of α and β, on the current topologic order, is computed, then only those topologic orders between these two locations require to be examined for possible changes, and the rest remain the same. Moreover, the heads of all operations located before the position of α and the tails of all operations located after the position of β remain unchanged.

Having discussed construction and point-based strategies, we now briefly review the related pool-based strategies. In these strategies, the search is guided by a pool of genotypes, each capable of being converted to a phenotype or solution through a decoder. These genotypes are changed in the coding space and their effects are evaluated on the phenotypes or solution space. The key point with these algorithms is that the genotypes which have led to higher quality phenotypes receive higher chances of surviving in the pool. In the literature of genetic algorithms, genotypes are referred to as genomes or chromosomes.

Two sets of operators called crossover and mutation are aimed at making changes to the genomes. Whereas crossover operators are binary operators and operate on two operands (genomes), mutation operators are unary operands and slightly perturb a genome on which they operate. A selection scheme, which favors solutions with higher quality, biases the process of the search towards generating a superb solution.

In the JSP, the genomes can be either a direct representation of the problem or an indirect one. As an example of direct representation, we can mention the completion times of operations (Yamada & Nakano, 1992), or any other mechanisms which determine the priorities of operations in seizing their corresponding machines (Della Croce et al., 1995; Dorndorf & Pesch,

1995). On the other hand, as an example of indirect representation we can mention the order of machines to be scheduled one after another by a one-machine problem (Dorndorf & Pesch, 1995).

The direct representation is not usually as powerful as the indirect one. Two nearly efficient algorithms which use direct representation, (Della Croce et al., 1995; Dorndorf & Pesch, 1995), both employ a pool of survived dispatching rules in constructing new solutions. Each genome is represented with *mn* genes, and shows the priorities of operations. The Giffler and Thompson's procedure has been used as a decoder to convert these genes to a solution.

In both algorithms, the crossover operator employed is uniform, in the sense that it selects priorities from each parent randomly either at the time of constructing a solution or at the time of creating an offspring. In both algorithms, the Giffler and Thomson method is used as a decoder, and in the second case the method operates as a repairing mechanism and removes the infeasibility which may have occurred in the offspring.

Whereas in the first algorithm, each genome represents operations based on their start time, in the second algorithm, each genome has been sub-grouped based on the machines their genes require. In (Bierwirth et al., 1996), crossover operators employed for permutation problems, in general, and for the JSP, in particular, have been examined and it has been found that the strongest phenotypical correlation between the parents and offspring occurs when they preserve absolute order, and not relative (side by side) or position order.

Another successful genetic algorithm has also been presented in the same paper (Dorndorf & Pesch, 1995), which uses indirect representation. As its genome structure, this algorithm, instead of selecting a sequence of dispatching rules, with the size of *mn*, considers a sequence of machines, with the size of *m*. It employs the shifting bottleneck procedure, as its decoder, which

converts a genome, as sequence of machines, to a solution, through scheduling the machines one after another by the SBP.

This algorithm owes its effectiveness mainly to the appropriate representation of genomes and using the SBP as its decoder. The effect of the representation and the decoder can become more apparent when we notice that it was the incorporation of the Carlier's method, as the base of the SBP, into an implicit enumeration that for the first time solved ft10 to optimality. The genetic algorithm mentioned, instead of performing implicit enumeration, uses evolution to guide the SBP, albeit with losing the guarantee of optimality.

In (Cheng et al., 1996, 1999), a comprehensive tutorial survey on genetic algorithm for the JSP has been presented. In effect, these two papers, as two parts of the same research, present an extensive and tutorial survey on the genetic algorithms presented for the JSP. In (Yamada & Nakano, 1992), a genetic algorithm has been presented which uses the completion times of activities as the genes of its employed genomes.

The crossover mechanism applied is founded on the Giffler and Thompson's algorithm for generating active schedules based on two sets of completion times of operations, as two genomes. Based on these two parent genomes, two offspring genomes are generated and among these four genomes two of them are selected as follows. First, the one which has the highest quality is selected. This genome is either one of the offspring or parent genomes. If it is a parent then among the two offspring genomes the one with the highest quality is selected. Otherwise, the other offspring genome is chosen and both of the offspring genomes go to the next generation.

The population size has not been specified explicitly and it seems that it is equal to two. Mutation ratio has been set to 0.01, implying that with the chance of 1 percent, the Giffler and

Thomson procedure selects the operation randomly. In other 99 percent of occasions, the operation is randomly selected from one of the two parents, with equal chance.

In the genetic algorithm presented in (Falkenauer & Bouffouix, 1991), the genomes show the order of operations on each machine and the crossover operator employed is applied independently to each part of the genome, based on its corresponding machine. In their research, several different crossover operators have been tested and the results have been compared with one another.

A hybrid genetic algorithm has been presented in (Gonçalves et al., 2005) in which the genomes are shown by random keys. In this procedure, each schedule is constructed through the use of priorities shown by these random keys, and this scheduling the scheduling is performed by the means of a parameterized method. The rationale of using a parameterized method is that whereas optimal solutions are in the set of active schedules, this set contains many schedules with large delays and this decreases the average quality of solutions existing in this set. The parameterized method controls the maximum delay times of each operation through a gene employed for the corresponding operation.

A pool-based procedure has been presented in (Yin et al., 2011) which is based on discrete artificial bee colony (Karaboga & Basturk, 2007). In the colony of bees, each food source is a permutation consisting of mn elements, with each job being repeated m times. The nectar of each food source shows the corresponding makespan. Bees are divided into three groups of *employed*, *onlooker*, and *scout*. Each employed bee is represented by a permutation of jobs, and also initially a half of the bees are employed, and the rest are onlooker.

Any employed bee which abandons its food source becomes a scout and starts looking for a new food source. The responsibility of an onlooker bee is to obtain information of food source, based

on their nectars, from the employed bees and to improve it. For the purpose of such improvement, based on a random biased sampling, one of the food sources is selected and initially mutated. The mutation operations used consist of swapping, insertion, and inversion. Among the mutation operations, one is randomly, but not uniformly, selected and applied. Then the onlooker bee applies a local search to the mutated food source with a certain probability. The local search is performed by repeatedly using one of the above three mutation operations for a large number of *mn(mn-1)* times. If a solution is not improved for a specified number of trials, this solution is abandoned by its corresponding employed bee, and the bee will become a scout and will search for a random food source. The solution obtained at the end of the procedure undergoes a pairwised-based local search in which the orders of two adjacent jobs on a machine are interchanged. It seems that this final part of the procedure has a crucial role in its efficiency.

The hybrid genetic algorithm presented in (Park et al., 2003) uses a single and a parallel genetic algorithm based on unpartitioned operation-based representation, described in (Bierwirth et al., 1996), which in general produces viable solutions. In this representation, each job is repeated *m* times along the corresponding encoding. In effect, in this representation, the k^{th} occurrence of each job shows its k^{th} operation of its technologic sequence. Giffler and Thompson algorithm has been used for both decoding and generating initial solutions.

The procedure presented in (Hasan et al., 2009) combines a genetic algorithm with a local search. In the GA employed, each individual represents a particular schedule and is shown with a binary chromosome. Since a JSP with *m* machines and n jobs has $(n!)^m$ possible solutions which are mainly infeasible, the employed chromosomes may show infeasible solutions. That is why the authors have used a repairing mechanism to remove such infeasibility.

In (Lin et al., 2010) an efficient pool-based procedure has been presented that combines genetic algorithms, particle swarm optimization, and simulated annealing. In this hybrid, a multi-type individual enhancement scheme has been used as the base of the employed local search. A random key representation has also been employed as the encoding scheme. The multi-type individual enhancement scheme consists of three operations of swapping, insertion, and inversion. The enhancement, which is performed to find better neighbors, does not use any information about the critical path at all. These three operations are done with their corresponding probabilities and change random keys on the encoding. A main point of this method is that despite not using any information about critical paths, it produces very good results.

The procedure presented in (Bierwirth, 1995) uses a generalized permutation approach; it avoids infeasibility by replacing an *m*-partitioned permutation with an unpartitioned one. In this single permutation, each job appears *m* times, i.e. the number of machines it requires. Then such single representation can be converted to *m* separate representations, each showing a feasible permutation on its corresponding machine. In comparison to the multiplication of the solution size of *m* permutations, such a single permutation has much larger solution size.

The procedure presented in (Nasiri & Kianfar, 2012) is a hybrid which combines elements from path relinking, tabu search, and global equilibrium to tackle the JSP. The three major components of this hybrid are (i) storing the history of search for better exploration of search space, (ii) reducing the distance between two processing orders gradually, and (iii) using an effective tabu list preventing any visit of previous solutions. Also, as its neighborhood structure, it uses a modified version of N6. Table 1 presents a summary of research background and literature review performed.

Table 1 is inserted here.

4. TGA

As an asexual genetic algorithm equipped with a tabu search, the TGA uses the notion of elite pool and selectively picks the solution with the highest quality from the pool. It also potentially removes the lowest quality solution from the pool before a new solution is added to the pool. Non-delay schedules generated by the Giffler and Thompson method fill the initial pool. In each iteration of the genome processing, the highest quality genome in the elite pool is removed and mutated in a biased fashion through running a limited tabu search procedure.

Exploitation has been balanced versus exploration by a decision strategy that mainly determines whether any intermediate solution along the run of tabu search should join the elite pool. This decision strategy also determines whether upon joining a new solution to the pool, the worst solution should remain in the pool. The process of adding genomes to the pool and deleting them from the pool is repeated until a time limit is reached or the elite heap becomes empty.

Calling the Giffler and Thompson (GT) procedure in the forward and backward fashion provides an opportunity to enhance the solutions provided by this procedure. In effect, by considering a mirror disjunctive graph, the direction has become flexible and the initial solution provided by the GT can be in either forward or backward direction. Whereas the term forward indicates that, in the solution procedure, the original disjunctive graph has been used, the term backward indicates that the mirror disjunctive graph has been employed. Compared to the original disjunctive graph, in the structure of the mirror disjunctive graph, all arcs have been reversed and their cost shows the duration of their new starting operation.

Depending on whether the initial solution has been generated by the forward or backward direction, the enhancement should be performed by using the other direction. Hence, because

the initial solutions have been generated in the forward direction, the backward direction has been used to potentially improve the result. Such a combination of forward and backward directions guarantees that a great deal of fine-tuning adjustments is performed and there is no significant computational burden left for further enhancement.

Whereas, the priority of each operation in the backward direction is its completion time determined in the forward direction, the priority of each operation in the forward direction is its starting time determined in the backward direction. Computational experiments show that solutions generated in this way have higher quality than solution generated in either forward or backward direction alone.

For instance, our computational experiments show that consistently the best solution among 2000 solutions in which 1000 solutions have been generated in the forward and the rest in the backward direction, has significantly lower quality than the best solution among 1000 solutions that each has been generated in one direction and improved in the opposite direction.

Switching from one direction to the other direction can continue until no improvement becomes possible. Figure 4 shows a solution obtained by the Giffler and Thompson procedure for a benchmark instance called ft06 (Fisher & Thompson, 1963), it also demonstrates how by applying backward operations we have been able to decrease the makespan from 64 to 58. In this figure, the numbers written on operations is the machine number the corresponding operation requires. As is seen, in the backward application of the Giffler and Thompson method, the operations one after another, based on their completion time in the forward application, have been pulled towards the opposite end of the chart, and the six units reduction, 64-58, in the makespan is the result of such pulling.

Figure 4 is inserted here.

In our experiments we noticed that there are rare occasions that such pulling leads to the degrading of solutions. Figure 5 shows that the running of forward/backward Giffler and Thompson method does not always improve the makespan. The JSP instance is again ft06 and the corresponding machine number has written on each operation.

Figure 5 is inserted here

As can be seen, the operation of job 3 on machine 1 is pulled towards left; however, at t=26, when machine 1 becomes idle, the operation of job 4 cannot be started (as it did on the forward schedule). The reason is that, at t=26, job 4 is being processed on machine 3 (scheduling backward). Hence machine 1 processes job 6 before job 4; this causes the starting time of job 4 on machine 2 to be delayed one unit and eventually makes the makespan one unit longer. This situation happened in less than 1% of times in our preliminary experiment (8 out of 1000 randomly generated schedules) for the instance ft06.

The forward-backward procedure can even be employed in an extended mode. In such a mode, the backward or forward directions are not tried once but for several times, say k times. The larger number of trials improves the chance of enhancing the solution because, for instance, if one backward direction is not able to make an improvement to the solution generated in the forward direction, using another forward-backward running may do so.

When k is greater than 1, the priority of critical operations can be determined randomly, with the chance of β, say $\beta = 2/3$, and determined the same as that of other operations with the chance of $(1- \beta)$. The rationale behind such a suggestion is that if the critical operations do not change their relative order, no improvement is expected. For the purpose of simplicity, the case of extended mode has not been employed in the current procedure and k has been set to 1.

The next point to discuss is the way in which the preciseness of estimates is traded off with the speed of estimation, emphasizing that the speed in which moves are evaluated has a decisive role in the efficiency of the procedure. For this purpose at the cost of losing insignificant precision, and losing the guarantee for producing a lower bound, our approximate procedure for producing an effective estimate of makespan works as follows.

Suppose that a critical block for the machine *i* consists of the following operations:

$$\varphi_i^{\alpha}, \varphi_i^{\beta}, \varphi_i^{\gamma}, \ldots . \varphi_i^{\varepsilon}, \varphi_i^{\theta}, \varphi_i^{\lambda}, \varphi_i^{\mu}, \ldots . \varphi_i^{\omega} \tag{8}$$

Now suppose that the operation φ_i^{λ} is moved before the operation φ_i^{α}. A chain is created as

$$\varphi_i^{\lambda}, \varphi_i^{\alpha}, \varphi_i^{\beta}, \varphi_i^{\gamma}, \ldots . \varphi_i^{\varepsilon}, \varphi_i^{\theta} \tag{9}$$

This chain has a machine successor and a machine predecessor. Its machine predecessor is $mp(\varphi_i^{\alpha})$ and its machine successor is φ_i^{μ}. It should be noticed that the case where an operation moves after the operation φ_i^{ω}or the cases where the operation φ_i^{α} or φ_i^{ω} moves in the middle of the block are exactly treated in the same way through a chain that has a machine predecessor and a machine successor. Based on the chain created and its machine successor and predecessor, the operations are performed as follows. First, the effect of moving the operation φ_i^{λ} before the operation φ_i^{α} is estimated as follows:

$$Eval = 0 \tag{10}$$

$$\overline{head}(\varphi_i^{\lambda}) = \max\left\{head\left(mp(\varphi_i^{\alpha})\right), head\left(jp(\varphi_i^{\lambda})\right)\right\} + t(\varphi_i^{\lambda}) \tag{11}$$

$$Eval = \max\{Eval, \overline{head}(\varphi_i^{\lambda}) + Tail(js(\varphi_i^{\lambda})\} \tag{12}$$

$$\overline{head}(\varphi_i^{\alpha}) = \max\left\{\overline{head}(\varphi_i^{\lambda}), head\left(jp(\varphi_i^{\alpha})\right)\right\} + t(\varphi_i^{\alpha}) \quad (13)$$

$$Eval = \max\{Eval, \overline{head}(\varphi_i^{\alpha}) + Tail(js(\varphi_i^{\alpha})\} \quad (14)$$

$$\overline{head}(\varphi_i^{\beta}) = \max\left\{\overline{head}(\varphi_i^{\alpha}), head\left(jp(\varphi_i^{\beta})\right)\right\} + t(\varphi_i^{\beta}) \quad (15)$$

$$Eval = \max\left\{Eval, \overline{head}(\varphi_i^{\beta}) + Tail(js(\varphi_i^{\beta})\right\} \quad (16)$$

In the same manner, the value of $Eval$ is updated for φ_i^{γ} , ... φ_i^{ε} , φ_i^{θ} ,one after another.

Finally the value of $Eval$ is computed as:

$$\overline{head}(\varphi_i^{\mu}) = \max\left\{\overline{head}(\varphi_i^{\theta}), head\left(jp(\varphi_i^{\mu})\right)\right\} + t(\varphi_i^{\mu}) \quad (17)$$

$$Eval = \max\{Eval, \overline{head}(\varphi_i^{\mu}) + \max\{Tail(js(\varphi_i^{\mu}), Tail(ms(\varphi_i^{\mu})\}\} \quad (18)$$

A main characteristic of this evaluation is that the result is not a lower bound for the makespan, in the sense that in some occasions it can be greater than the makespan. It could become a lower bound by setting to zero both (i) the head of any operation whose starting time is smaller than that of the operation φ_i^{α} , and (ii) the tail of any operation whose starting time is smaller than that of the operation φ_i^{λ} .

Forcing it to be a lower bound, however, can degrade the quality of this estimate and consequently decreases the quality of the obtained solutions. Based on computational experiments performed, this fast estimate in 79.8% of cases exactly shows the value of the makespan, in 20.1% of cases is lower than the makespan and in only 0.05% percent of cases is greater than the makespan.

The value of $Eval$ can also be further updated with the following equations:

$$\overline{Tail}(\varphi_i^{\theta}) = \max\left\{Tail\left(js(\varphi_i^{\theta})\right), Tail(\varphi_i^{\mu})\right\} + t(\varphi_i^{\theta}) \quad (19)$$

$$Eval = \max\{Eval, \overline{Head}(\varphi_i^{\theta}) + \overline{Tail}(\varphi_i^{\theta}) - t(\varphi_i^{\theta})\} \quad (20)$$

$$\overline{Tail}(\varphi_i^{\varepsilon}) = \max\left\{Tail\left(js(\varphi_i^{\varepsilon})\right), \overline{Tail}(\varphi_i^{\theta})\right\} + t(\varphi_i^{\varepsilon}) \quad (21)$$

$$Eval = \max\{Eval, \overline{Head}(\varphi_i^{\varepsilon}) + \overline{Tail}(\varphi_i^{\varepsilon}) - t(\varphi_i^{\varepsilon})\} \quad (22)$$

In the same manner, the value of $Eval$ is updated for all operations before φ_i^{ε} ,one after another, up to the operation φ_i^{α} , for which the update is the same as that of other operations:

$$\overline{Tail}(\varphi_i^{\alpha}) = \max\left\{Tail\left(js(\varphi_i^{\alpha})\right), \overline{Tail}\left(\varphi_i^{\beta}\right)\right\} + t(\varphi_i^{\alpha}) \quad (23)$$

$$Eval = \max\{Eval, \overline{Head}(\varphi_i^{\alpha}) + \overline{Tail}(\varphi_i^{\alpha}) - t(\varphi_i^{\alpha})\} \quad (24)$$

Finally the value of $Eval$ is computed as:

$$\overline{Tail}(\varphi_i^{\mu}) = \max\{Tail(js(\varphi_i^{\mu}), \overline{Tail}(\varphi_i^{\alpha})\} + t(\varphi_i^{\mu}) \quad (25)$$

$$Eval = \max\{Eval, \overline{Head}(\varphi_i^{\mu}) + \overline{Tail}(\varphi_i^{\mu}) - t(\varphi_i^{\mu})\} \quad (26)$$

As is seen in these equations only has the updated values of $Head$ been used, shown with $\overline{Head}$. The reason is that in the previous equations the values of $Head$ were updated for all $\varphi_i^{\alpha}, \varphi_i^{\beta}, \varphi_i^{\gamma}, \ldots . \varphi_i^{\varepsilon}, \varphi_i^{\theta}, \varphi_i^{\lambda}$, and φ_i^{μ}.

Again, the result is not a lower bound for the makespan. It could become a lower bound by setting the tail of any operation whose starting time is smaller than that of the operation φ_i^{α} to zero. It should be noticed that since instead of heads ($Head$) the updated head values ($\overline{Head}$) have been used, for keeping the value of $Eval$ as a lower bound they need no modification. The same in the computation of heads, making changes to tails for guarantying that $Eval$ is a lower bound of the makespan degrades the quality of the obtained solutions. Based on our final

computational experiments, we decided not to use this second part of the computation in the procedure and only has the first part been activated in the final procedure.

For instance, in the given block, if instead of φ_i^{λ}, which moved before φ_i^{α}, any other operation had moved before or after any other operation, only would the chain of operations have changed, with its corresponding machine predecessor and machine successor. In effect, all calculations for this machine predecessor, chain, or machine successor would remain the same.

In line with (Balas & Vazacopoulos, 1998; Pardalos & Shylo, 2006), the following method has been used to make the calculation of topologic order fast. Assuming that a single operation of a block has moved from its current location to some other location of the block, the updating of the topologic order is performed recursively as follows.

Assuming that this change has affected only the locations between *x* and *y* of the topologic order, with *y* greater than *x*, there are two options, (i) the moved operation has originally been in location *x*, (ii) the moved operation has originally been in location *y*. In the first case, only the successors of *x* located in the range between *x* and *y* are determined recursively and moved with *x*, whereas in the second only the predecessors of *x* are determined recursively and moved with *x*.

For the determination of the successors in the range between *x* and *y*, the procedure finds the immediate successors of the moved operation and any of them which are in the range between *x* and *y* is recorded and its immediate successors in the range between *x* and *y* are found. This continues recursively until all successors of the moved operation are scanned.

The recursive determination of the predecessors is the same as that of the successors and the only difference is that instead of immediate successors, the immediate predecessors are considered. For example, consider the following topologic order of operations.

…….2,6,8,3,1,9,4,5,7……

Assuming that the operation 2 is supposed to move after the operation 5, then the operation 2 takes with itself all its successors in the range between 2 and 5. If the operations 1, 3, and 4 are the successors of the operation 2, then the new topologic order is as follows.

…….6,8,9,5,2,3,1,4,7……

A point with these successors is that they keep their relative order when they move. For instance, as is seen, operation 1 has appeared after operation 3 and operation 4 has appeared after operation 1.

The complete description of the TGA, presented in Figure 6, is as follows. First, lines 3 and 4 generate initial solutions. For this purpose, line 3 generates *n* solutions with the randomized Giffler and Thompson (RGT) heuristic. In each step of the RGT, from the top *K* jobs, which have been prioritized based on their remained processing times, a random job is selected, with the parameter *K* basically controlling the diversity of initial solutions generated. Then among those *n* solutions generated, line 4 selects the top *m* solutions and fills the pool with these *m* solutions.

Figure 6 is inserted here

The main loop of the pseudocode starts at line 5. With this loop, in each iteration, a number of statements shown between lines 6 and 41 are executed and aim to make possible improvements in the current best solution obtained. First, at line 7, the best element is removed from the pool and the current solution, *x*, is set to it. Then, at line 8, with a chance of *p1,* the current solution, x, undergoes a *mutation.* This mutation which is based on the *N1* neighborhood simply swaps two adjacent operations on a random block of operations on the critical path.

Next, in lines 9-14, with a chance of *p2,* the Backward Giffler and Thompson (BGT) procedure is run on the current solution, *x*. The BGT, as a backward procedure, is a mirror of the Giffler and Thompson procedure, and runs the original procedure with the assumption that the

operations of each job should be processed in the reverse direction. The reason for applying the BGT is that in some occasions, applying a backward procedure on a solution obtained with a forward procedure can lead to improving the solution.

Since the application of the BGT, as was shown in the previous sections, can also deteriorate a solution, line 13 prevents such possible deterioration by only allowing a revised solution, y, which is of higher quality to replace the current solution, x. In other words, if the resulting solution can improve the current solution, it replaces the current solution; otherwise, the current solution remains intact.

In lines 15-40, as a facilitating routine, a limited tabu-search (LTS) is performed on the current solution and during the execution of this tabu-search, any possible high-quality solution obtained can join the pool of solutions. In the implementation of this tabu search, each element of the corresponding tabu list shows a sequence of operations on a particular machine. Hence, in traversing the neighborhood graph, a move is considered tabu if it results in a schedule which has an identical sequence of operations to an element in the tabu list with respect to a particular machine.

To enhance the exploration power of the procedure, line 15 sets the size of the tabu list to a uniformly random number between 10 and 14 and lines 20 and 21 alternate the neighborhood structure between N5 and N6′ based on the *N5N6'Probability* parameter. By N6′ neighborhood we mean an extension of N6 neighborhood in which either the starting or the ending operation can also move to the other part of block, if feasible. Hence N6′ neighborhood can generate four groups of permutations. These groups are respectively created by inserting (i) each operation before the first operation, (ii) each operation after the last operation, (iii) the first operation right after each other operation, and (vi) the last operation right before each other operation.

Lines 22 through 35 are aimed at determining the next neighbor to move to. If the best candidate neighbor is of higher quality than the best solution in the current round of the tabu search, line 34, based on the employed *aspiration criteria* and regardless of whether a neighbor is tabu, selects such a neighbor. Otherwise, the best non-tabu neighbor is selected. In rare occasions, all the neighbors may become tabu. Line 35 takes care of this possible case and after perturbing the best solution in the current round of the tabu search, based on *N1* neighborhood, selects this perturbed solution as the neighbor to which the move is done.

Line 39 manages to conditionally put high quality solutions generated in the process of the tabu search in the pool. It operates as follows. If the pool is full and the solution is of higher quality than the worst solution of the pool, it replaces the worst solution. However, if the pool is not full, the current solution joins the pool with a probability of $p = e^{\frac{-L}{\lfloor \theta . \log_2 f \rfloor}}$, where L is the distance to the current best solution, θ is a parameter controlling the openness of the pool, and f is equal to $\frac{s}{c}$, where s and c are the pool size and pool capacity, respectively. Moreover, because of diversification purposes, whenever a new solution joins the pool, for the next k iterations, no solution can join the pool. Based on the preliminary experiments, the value of k has been set to 5. Figure 7 shows the summary of steps involved in TGA. Also, Figure 8 shows how the application of the pseudocode provides, in its very first iterations, the optimal solution of the sample problem of Figure 1.

Figures 7 and 8 are inserted here

As is seen, the TGA is an evolutionary procedure with an asexual reproduction mechanism. Figure 9 shows how the frequency of solutions in the pool changes with respect to the makespan and distance to the optimal solution. As the figure shows, the convergence to optimal solution for the famous ft10 instance happens in less than 0.1 seconds.

Figure 9 is inserted here

5. Computational Experiments

The proposed procedure has been run on a selection of well-known benchmark instances extracted from ORLIB site managed by Brunel University, UK. This site which includes most of the benchmark instances produced for operational research problems is the well-known repository of the JSP benchmark instance.

These instances belong to six different categories, namely (i) 3 classic instances from (Fisher & Thompson, 1963) named *ft06*, *ft10*, and *ft20,* (ii) 40 instances from (Lawrence, 1984) named *la01-la40 ,* (iii) 5 instances *abz5-9* from (Adams et al., 1988), (iv) 10 instances *orb01-10* from (Applegate & Cook, 1991), (v) 4 instances *yn1-4* from (Yamada & Nakano, 1992), and (vi) 10 instances, *swv01-10* from (Storer et al., 1992).

The TGA has been implemented in C++ and compiled via GNU GCC compiler on a DELL PC with 2.2 GHz speed and 8 GBs of memory. The parameters of the procedure having been set based on Table 2. Moreover, for each instance, in line with other similar procedures, the TGA has been run for 10 times with a time limit of τ seconds for each run. The value of τ has been determined based the formula of $\max\{1\,,\,\frac{n}{m}.(9n-60)\}$ seconds. Moreover, with taking the value of the optimal solution as input, if it exists, the procedure can stop as soon the optimal makespan is found.

Table 2 is inserted here.

Table 3 shows the performance of the TGA on 43 instances and compares the procedure with one of the fastest available procedures for the JSP called Tabu search-Simulating Annealing,

TSSA presented in (Chao Yong Zhang et al., 2008). It is worth mentioning that by its authors the running times of the TSSA have been reported on a Pentium IV 3.0 Ghz CPU.

Table 3 is inserted here.

The size of each instance (*number_of_jobs* × *number_of_machines*) has been written in the second column of Table 3. The best known Lower Bound (LB), and the best known Upper Bound (UB) available in the literature have also been presented in columns 3 and 4, respectively. These upper and lower bounds have been extracted from (Jain & Meeran, 1999) and (Chao Yong Zhang et al., 2008) and in the cases where LB is equal to UB (BKS), the BKS is the optimal solution of the problem instance. In Table 3 the column named as $Best$ provides the best makespans returned by the TGA. The columns named as T_{best}, Avg, and T_{Avg} provide the minimum time needed to reach the best makespan in seconds, the average of makespan in 10 runs, and the average time for 10 runs in seconds, respectively.

As seen in Table 3, for small instances the TGA is very fast. For example, the instance ft10 has been solved in as small as 0.668 second, on average. In effect, for this instance, the TGA performs more than five times faster than the TSSA. The table also shows that for 26 out of 43 instances the procedure has found the best available solutions in the literature.

Since Chao Yong Zhang et al. (2008) have not reported the results on all 40 instances due to Lawrence (1984), TGA performance on the remaining 29 instances has been presented in Table 4. As can be seen, TGA finds the optimal solutions for all instances in a less than 0.2 second.

Table 4 is inserted here.

6. Concluding Remarks

Evolutionary approaches are established based on two distinct principals; the first principal is involved with finding the characteristics of superb solutions and spreading these characteristics in the entire population, and the second one is the survival of the fittest principle. A vital characteristic of local searches which makes them similar to the evolutionary processes is that local searches alternate between coding space (genotype), and solution space (phenotype). In local searches, similar to evolutionary searches, whereas modifications are carried out on the coding space, the evaluation takes place on the solution space.

Genotype-phenotype duality used in both local and evolutionary searches makes the integration of these two searches seamless. Since search effectiveness, regardless of local or evolutionary type, is characterized by two extreme conflicting factors of exploration and exploitation, successful searches are those which seamlessly integrate the two paradigms of evolutionary and local searches in the direction of a trade-off between exploration and exploitation.

By using the notion of elite pool and picking the solution with the highest quality from the pool and performing a local search on such a solution, the TGA integrates evolutionary and local searches towards delicately balancing exploration versus exploitation. It is such delicate balance that has enabled the procedure to obtain solution values with the precision of one percent from the best available ones in the literature in a matter of seconds.

Two intuitive factors in measuring the effectiveness of the TGA are its coverage of the search space and its intensification on exploring high-quality solutions. These two measures are, however, two highly conflicting matters because the more it concentrates on covering larger areas of the search space, the less it can capitalize on searching the neighbors of the encountered high quality solutions.

The current balance between these two conflicting measures can be enhanced by the further study of the effect of different components of the TGA on its performance. For instance, converting non-delay schedules to active schedules or making the mutation aggressive can shift the balance towards diversification. On the other hand, permitting larger number of intermediate solutions, along the run of tabu search, to join the elite pool and removing the worst solution upon joining a new solution to the pool shifts the balance towards intensification.

The major point which should be considered in such possible further study is that it is not the fine-tuning of one component of the TGA that can significantly affect its performance but the synergetic matching of all components in the simultaneous increasing of the coverage, mobility and exploitation power of the procedure. This can be explained by the fact that different procedures in the literature which appear to be only superficial variation of one another can exhibit considerably dissimilar performance.

The issue is not simply that a component which is highly ineffective in its integration with some other components becomes highly effective when it is used by some other components. The deeper issue is that balancing intensification versus diversification is highly delicate and problem-specific. An improved version of the TGA can circumvent this issue by shifting the balance towards more exploitation and less exploration or vice versa adaptively. Such adaptation can be performed based on a feedback received from the fitness landscape, and is of paramount importance.

References

Adams, J., Balas, E. & Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. Management science, 34(3), 391-401.

Applegate, D. & Cook, W. (1991). A computational study of the job-shop scheduling problem. ORSA journal on computing, 3(2), 149-156.

Balas, E. & Vazacopoulos, A. (1998). Guided local search with shifting bottleneck for job shop scheduling. Management science, 44(2), 262-275.

Bierwirth, C. (1995). A generalized permutation approach to job shop scheduling with genetic algorithms. OR Spectrum, 17(2), 87-92.

Bierwirth, C., Mattfeld, D. & Kopfer, H. (1996). On permutation representations for scheduling problems. Parallel Problem Solving from Nature—PPSN IV, 310-318.

Bierwirth, C., Mattfeld, D. & Watson, J.P. (2004). Landscape regularity and random walks for the job-shop scheduling problem. Evolutionary Computation in Combinatorial Optimization, 21-30.

Carlier, J. (1982). The one-machine sequencing problem. European Journal of Operational Research, 11(1), 42-47.

Cheng, R., Gen, M. & Tsujimura, Y. (1996). A tutorial survey of job-shop scheduling problems using genetic algorithms--I. Representation. Computers & Industrial Engineering, 30(4), 983-997.

Cheng, R., Gen, M. & Tsujimura, Y. (1999). A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. Computers & Industrial Engineering, 36(2), 343-364.

Dell'Amico, M. & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. Annals of Operations Research, 41(3), 231-252.

Della Croce, F., Tadei, R. & Volta, G. (1995). A genetic algorithm for the job shop problem. Computers & Operations Research, 22(1), 15-24.

Dorndorf, U. & Pesch, E. (1995). Evolution based learning in a job shop scheduling environment. Computers & Operations Research, 22(1), 25-40.

Falkenauer, E. & Bouffouix, S. (1991). A genetic algorithm for job shop. In: (pp. 824-829 vol. 821): IEEE.

Fisher, H. & Thompson, G.L. (1963). Probabilistic learning combinations of local job-shop scheduling rules. Industrial scheduling, 225-251.

Giffler, B. & Thompson, G.L. (1960). Algorithms for solving production-scheduling problems. Operations Research, 8(4), 487-503.

Gonçalves, J.F., Mendes, J.J. & Resende, M.c.G. (2005). A hybrid genetic algorithm for the job shop scheduling problem. European Journal of Operational Research, 167(1), 77-95.

Hasan, S.M.K., Sarker, R., Essam, D. & Cornforth, D. (2009). Memetic algorithms for solving job-shop scheduling problems. Memetic Computing, 1(1), 69-83.

Jain, A.S. & Meeran, S. (1999). Deterministic job-shop scheduling: Past, present and future. European Journal of Operational Research, 113(2), 390-434.

Jain, A.S., Rangaswamy, B. & Meeran, S. (2000). New and "stronger" job-shop neighbourhoods: A focus on the method of Nowicki and Smutnicki (1996). Journal of Heuristics, 6(4), 457-480.

Karaboga, D. & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization, 39(3), 459-471.

Lawrence, S. (1984). Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). In: Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Lin, T.L., Horng, S.J., Kao, T.W., Chen, Y.H., Run, R.S., Chen, R.J., Lai, J.L. & Kuo, I. (2010). An efficient job-shop scheduling algorithm based on particle swarm optimization. Expert Systems with Applications, 37(3), 2629-2636.

Nasiri, M.M. & Kianfar, F. (2012). A GES/TS algorithm for the job shop scheduling. Computers & industrial engineering, 62(4), 946-952.

Nowicki, E. & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. Management science, 42(6), 797-813.

Nowicki, E. & Smutnicki, C. (2005). An advanced tabu search algorithm for the job shop problem. Journal of Scheduling, 8(2), 145-159.

Pardalos, P.M. & Shylo, O.V. (2006). An algorithm for the job shop scheduling problem based on global equilibrium search techniques. Computational Management Science, 3(4), 331-348.

Park, B.J., Choi, H.R. & Kim, H.S. (2003). A hybrid genetic algorithm for the job shop scheduling problems. Computers & industrial engineering, 45(4), 597-613.

Storer, R.H., Wu, S.D. & Vaccari, R. (1992). New search spaces for sequencing problems with application to job shop scheduling. Management science, 38(10), 1495-1509.

Taillard, E.D. (1994). Parallel taboo search techniques for the job shop scheduling problem. ORSA journal on computing, 6(2), 108-117.

Van Laarhoven, P.J.M., Aarts, E.H.L. & Lenstra, J.K. (1992). Job shop scheduling by simulated annealing. Operations Research, 40(1), 113-125.

Werner, F. & Winkler, A. (1995). Insertion techniques for the heuristic solution of the job shop problem. Discrete Applied Mathematics, 58(2), 191-211.

Yamada, T. & Nakano, R. (1992). A genetic algorithm applicable to large-scale job-shop problems. Parallel problem solving from nature, 2, 281-290.

Yin, M., Li, X. & Zhou, J. (2011). An efficient job shop scheduling algorithm based on artificial bee colony. Scientific Research and Essays, 5(24), 2578-2596.

Zhang, C.Y., Li, P., Rao, Y. & Guan, Z. (2008). A very fast TS/SA algorithm for the job shop scheduling problem. Computers & Operations Research, 35(1), 282-294.

Zhang, C.Y., Li, P.G., Guan, Z.L. & Rao, Y.Q. (2007). A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. Computers & Operations Research, 34(11), 3229-3242.

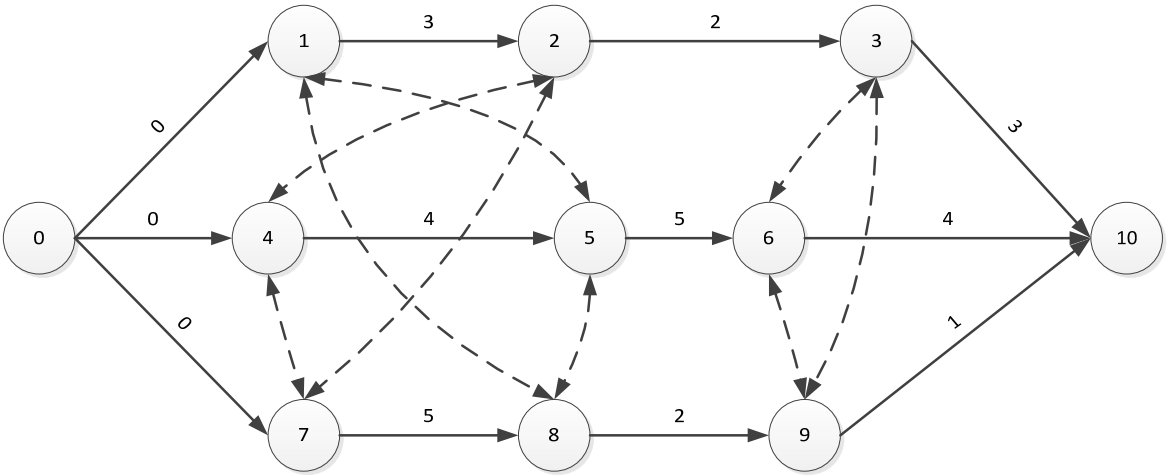


Figure 1.The disjunctive graph of a 3-job 3-machine JSP

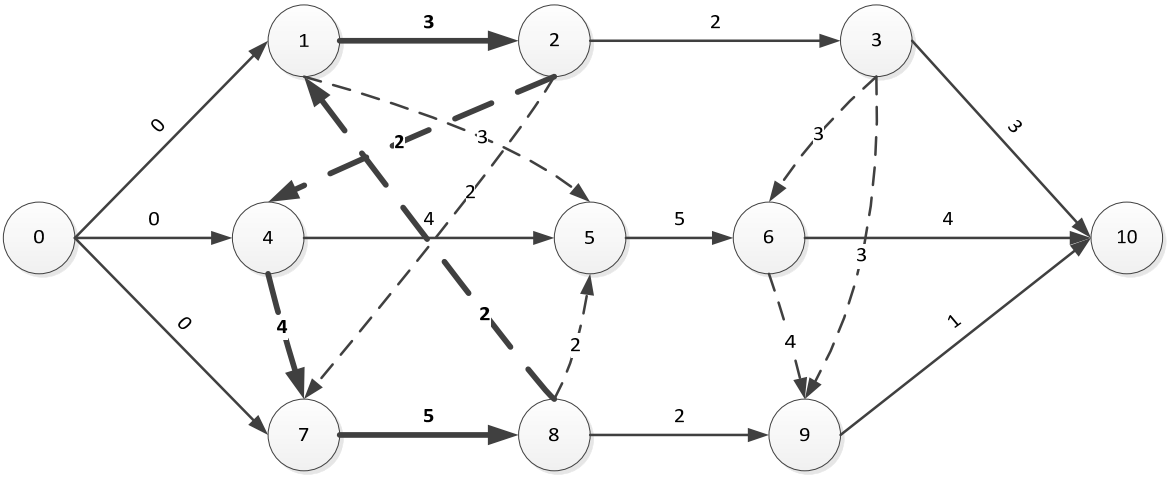


Figure 2. An infeasible solution which creates a loop (the loop has been shown in bold lines)

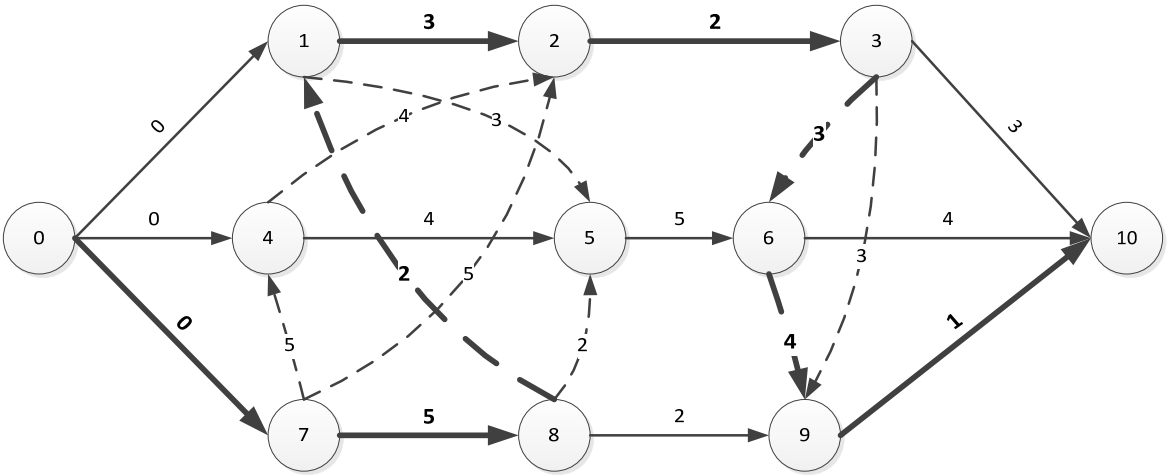


Figure 3. A sample solution with the makespan value of 20 (the critical path is shown in bold lines)

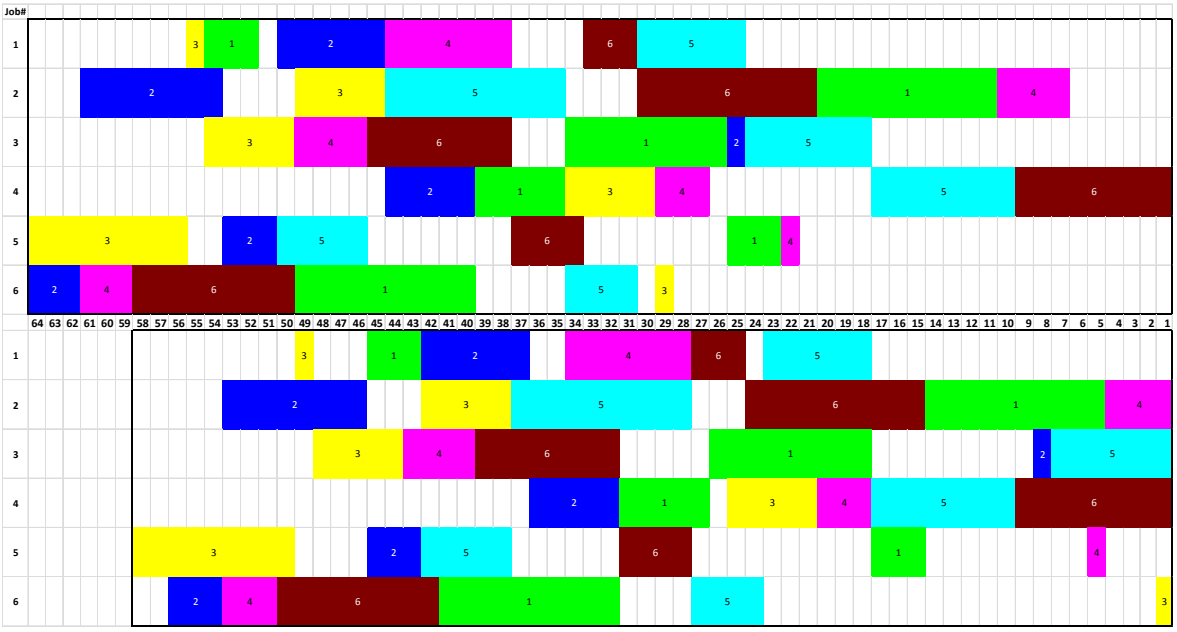


Figure 4. Showing how 6 units reduction in the makespan occurs by applying the backward Giffler and Thompson method on a solution obtained by the original method

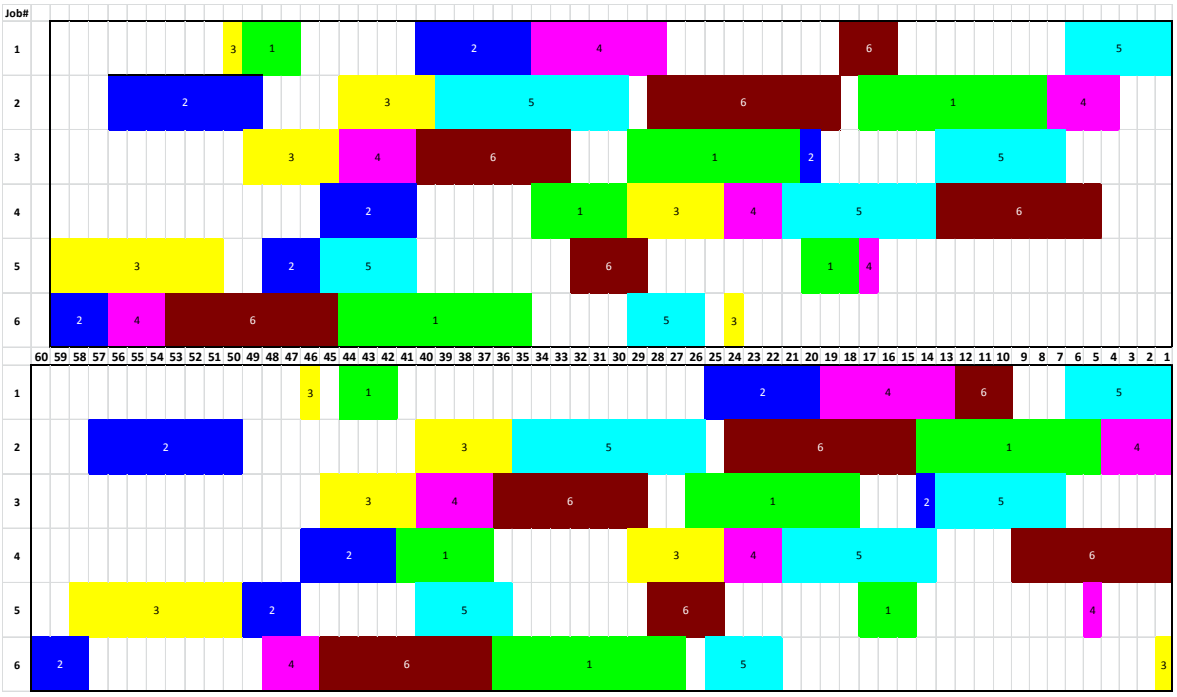


Figure 5. Showing an occasion that applying the backward Giffler and Thompson method on a solution obtained by the original method has led to increasing the makespan by one unit, from 59 to 60

```
01 Procedure TGA (n,m,p1,p2,minSizeTabu,maxSizeTabu,N5N6Prob,K)
02 {
03   Generate n solutions with Giffler and Thompson method;
04   Select the best m solutions and put them in the pool;
05   while (time-limit has not been reached and the pool has at least one element)
06   {
07       Take the smallest element out of the pool and call it x;
08       Mutate x with the chance of p1 with n1 neighborhood;
09       if (rnd < p2)//with the chance of p2
10       {
11            Apply Backward Giffler and Thompson method on the start times of x
12                                              and call the produced solution y;
13            if (the makespan of y is smaller than that of x) then set x to y;
14       }
15       Set tabu list size to a number between minSizeTabu and maxSizeTabu randomly;
16       Set tabu list to null;
17       while(the number of iterations with no improvement is less than MaxIter)
18       {
19          Find a critical path associated with x;
20           With the chance of N5N6'Prob generate neighbors with N5 neighborhood;
21           if N5 neighborhood was not used then use N6' neighborhood to generate neighbors,
22            Evaluate all neighbors with the fast estimate method;
23            Sort all neighbors based on their estimated makespan ascending;
24            i=1; //index of sorted neighbors
25            while(i≤K or all examined neighbors have been tabu)
26            {
27                   Take the i^th sorted neighbor and call it s;
28                   Calculate the exact value of the makespan of s;
29                    if s is the best neighbor in this loop call it y;
30                    if s is the best non-tabu neighbor in this loop call it z;
31                    i=i+1;
32                    if (i > number of neighbors) break; //no non-tabu neighbor
33            }
34           if (y meets aspiration criterion) set x=y;
35           else if (i > number of neighbors) perturb y with n1 and set x=y;
36           else set x=z; //at least one non-tabu neighbor has been encountered
37           //updating phase:38           Update heads, tails, and topologic orders based
on updated x
39           if (x meets the criterion of joining to pool) add x to pool;
40       }//end while(the number of iterations. . .
41   }//end while (time-limit has not been reached . . .
42 }//end of procedure
```

Figure 6. The c-type pseudo-code of the TGA

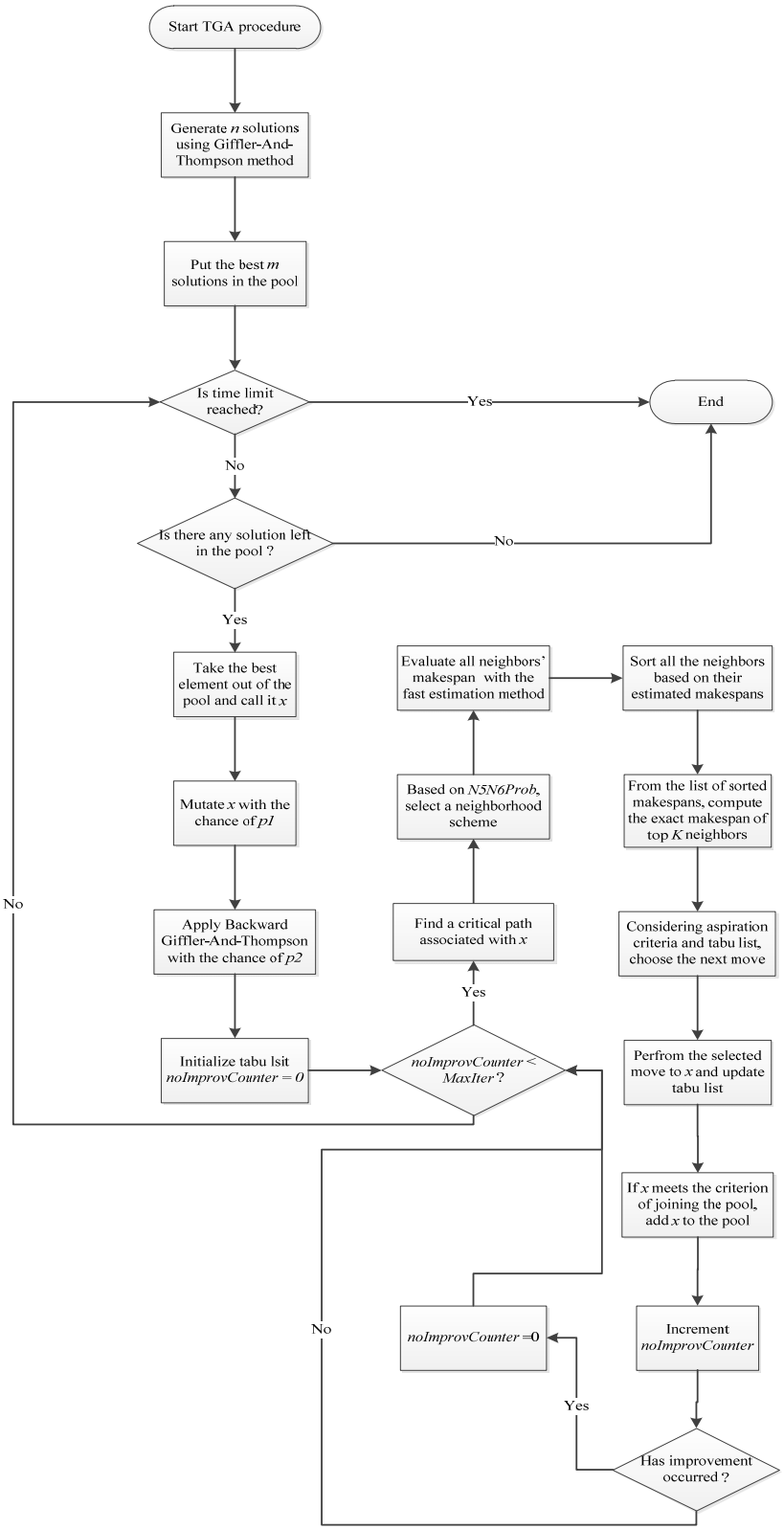


Figure 7. The flowchart of the TGA procedure

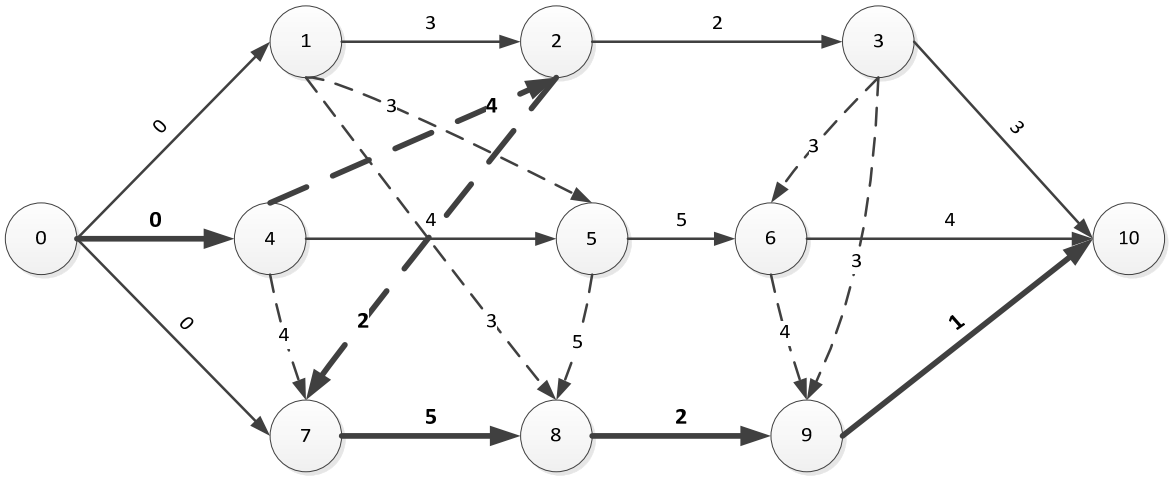
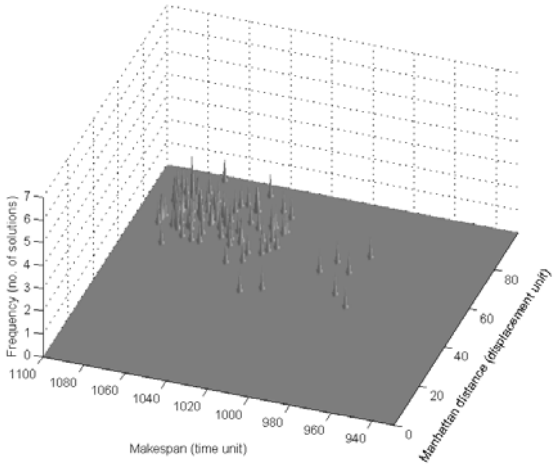
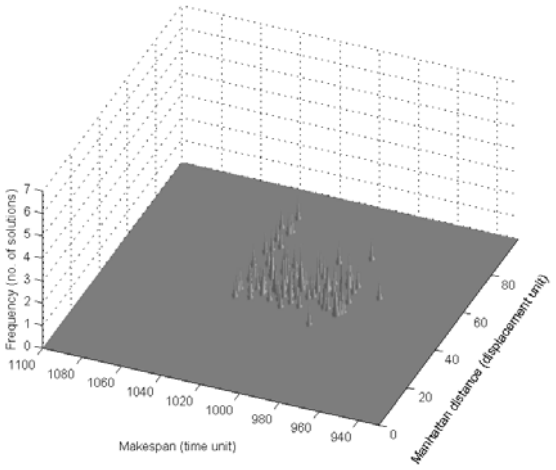


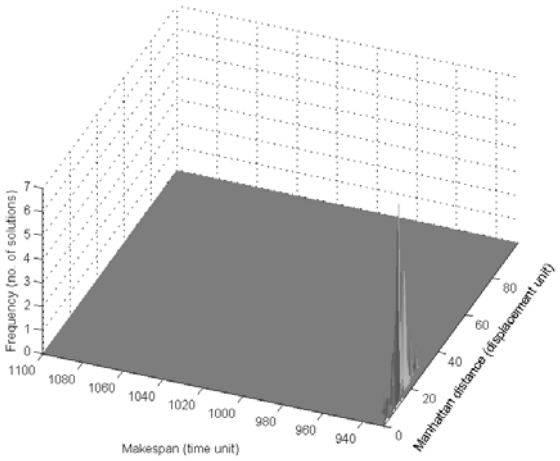
Figure 8. The optimal solution with the makespan of 14 (a critical path is shown in bold lines)


Frequency (no. of solutions)
0
1
2
3
4
5
6
7
1100
1080
1060
1040
1020
1000
980
960
940
Makespan (time unit)
0
20
40
60
80
Manhattan distance (displacement unit)

(a)



(b)



(c)

Figure 9. The distribution of solution in the pool of solutions for (a) the Initial stage, (b) middle stage (after running the algorithm for 0.01 seconds, and (c) final Stage, after 0.06 second, showing the final convergence of the pool around the optimal solution (930) for the instance ft10

Table 1. The summary of research background and literature review

Year	Authors	Summary
1960	Giffler and Thompson	generating active schedules, or their high quality subset called non-delay schedules based on the option selected by of users
1982	Carlier	developing a highly effective one-machine scheduling method for jobs with head, tail, and processing time
1988	Adams, Balas et al	sequencing the machines by the Carlier one-machine scheduling method through a Shifting Bottleneck Procedure (SBP)
1992	Van Laarhoven,	developing the famous N1 neighbourhood and employing it in a simulating annealing algorithm

	Aarts et al.	
1992	Yamada and Nakano	presenting a genetic algorithm with direct representation in which the genes show the completion times of operations, and Giffler and Thompson's algorithm is used for generating active schedules based on the two sets of completion times of operations, as two parents.
1993	Dell'Amico and Trubian	using both forward and backward insertion one after another and fixing the disjunctive arcs of the disjunction graph with the order in which operations are built into the schedule being based on the so called least-worsening rule
1995	Werner and Winkler	applying insertion techniques combined with a beam search which iteratively generate paths in a particular neighbourhood graph
1995	Dorndorf and Pesch	presenting a genetic algorithm with indirect representation in which the order of machines to be scheduled one after another by a one-machine problem is used as a genome
1996	Nowicki and Smutnicki	developing N5 neighbourhood, which interchanges the first two or the last two operations of each block with the exceptions of the first and last blocks
1998	Balas and Vazacopoulos	developing N6 neighbourhood, which moves each operation of a block after the last or before the first operation of the block, if feasible
2000	Jain, Rangaswamy et al.	analysing why the algorithm developed by Nowicki and Smutnicki (1996) is effective by fully examining its seven major components
2004	Bierwirth, Mattfeld et al.	analyzing the fitness landscape of the JSP and showing that such landscape, unlike those of many other combinatorial optimization problems, is non-regular ,in the sense of being biasedly connected
2005	Nowicki and Smutnicki	exploiting big valley property of landscape to develop an effective tabu search procedure which uses some elements of path relinking
2005	Gonçalves, Mendes et al.	presenting a genetic algorithm in which the genomes are shown by random keys, with a parameterized method controlling the maximum delay times of each operation through a gene employed for the corresponding operation
2007	Zhang, Li et al.	Developing an efficient move evaluation strategy which makes the calculations comparatively fast
2009	Hasan, Sarker et al.	combining a genetic algorithm with a local search and using a repairing mechanism to remove infeasibility in genomes.
2010	Lin, Horng et al.	Using random key representation and combining genetic algorithms, particle swarm optimization, and simulated annealing
2011	Yin, Li et al.	presenting a pool-based procedure which is based on discrete artificial bee colony, with three mutation operations of swapping, insertion, and inversion playing key roles
2012	Nasiri and Kianfar	combining elements from path relinking, tabu search, and global equilibrium

Table 2. The values set for the parameters of the procedure

Parameter	Value	Description
initSolGenerated	$nm + 400$	The number of initial solutions generated.
poolSize	100	The size of the pool of Elite Solutions
graspCardGT	6	The grasp cardinality of giffler and Thompson procedure.
backwardGTProb	60 %	Backward Giffler and Thompson is run with this chance.
perturbProb	$(20 * n/m)$ %	A solution is perturbed with this chance after being removed from pool.
opennessRatio	$14 + n/m$	This ratio, as described in section 4, is the value of θ in the formula $e^{\frac{-L}{\lfloor \theta . \log_2 f \rfloor}}$, which is the chance for joining pool of elite solutions.
lengthPrevented	$\lfloor 4 + n/m \rfloor$	Whenever a solution joins the pool, for the next length Prevented times no solution will join the pool.
Tabu Search Parameters:		
TotIterNonImprov	$\lfloor 2000 * n/m \rfloor$	When there is no improvement for this number of iterations, the tabu search procedure is halted.
MinTabuListSize	$\lfloor 14 + n/m \rfloor$	Tabu List Size Minimum value
MaxTabuListSize	$\lfloor 19 + n/m \rfloor$	Tabu List Size Maximum value
N5N6Prob	10 %	With this chance N4 scheme is chosen over N6' scheme.
ConsiderTripleMoves	60 %	With this chance triple moves are considered in tabu search.

Table 3. The results of comparing the TGA with the procedure presented in Zhang et. al. (2008) named as TSSA

				TGA				TSSA		
Instance	**Size**	**LB**	**BKS**	Best	Tbest	Avg	Tavg(s)	Best	Avg	Tavg
ft06	6 × 6	55	55	55	0.000	55	0.0	--	--	--
ft10	10 × 10	930	930	930	0.060	930	0.7	930	930	3.8
ft20	20 × 5	1165	1165	1165	1.610	1165	92.1	--	--	--
la19	10 × 10	842	842	842	0.025	842	1.2	842	842	0.5
la21	15 × 10	1046	1046	1046	10.775	1049.7	26.5	1046	1046	15.2
la24	15 × 10	935	935	935	0.200	935.3	51.2	935	936.2	19.8
la25	20 × 10	977	977	977	0.330	979	17.1	977	977.1	13.8
la27	20 × 10	1235	1235	1235	0.950	1236	45.3	1235	1235	11.7
la29	20 × 10	1152	1152	1153	6.370	1166.6	22.5	1153	1159.2	63.9
la36	15 × 15	1268	1268	1268	0.570	1268	5.1	1268	1268	9.9
la37	15 × 15	1397	1397	1397	0.510	1399.5	14.8	1397	1402.5	42.1
la38	15 × 15	1196	1196	1196	1.250	1197.7	5.7	1196	1199.6	47.8
la39	15 × 15	1233	1233	1233	0.500	1233.7	6.2	1233	1233.8	28.6
la40	15 × 15	1222	1222	1224	0.860	1225.5	11.1	1224	1224.5	52.1
abz5	10 × 10	1234	1234	1234	0.040	1234.6	11.5	--	--	--
abz6	10 × 10	943	943	943	0.030	943	0.1	--	--	--
abz7	20 × 15	656	656	659	1.130	664.9	50.7	658	661.8	85.9
abz8	20 × 15	645	665	667	1.480	674.6	38.2	667	670.3	90.7
abz9	20 × 15	661	678	678	3.250	686.7	33.9	678	684.8	90.2
orb01	10×10	1059	1059	1059	0.060	1060.8	5.6	1059	1059	3.5
orb02	10×10	888	888	888	0.060	888	0.7	888	888.1	6.4
orb03	10×10	1005	1005	1005	0.150	1010.7	1.4	1005	1012.5	13.8
orb04	10×10	1005	1005	1005	0.450	1010.6	4.8	1005	1008.3	14.3
orb05	10×10	887	887	887	0.760	887	10.0	887	888.6	6.6
orb06	10×10	1010	1010	1010	0.720	1012.2	8.6	1010	1010	8.5
orb07	10×10	397	397	397	0.020	397	0.1	397	397	0.5
orb08	10×10	899	899	899	0.090	899.7	5.8	899	902.5	7.2
orb09	10×10	934	934	934	0.090	936.7	2.6	934	934	0.4
orb10	10×10	944	944	944	0.030	944	0.1	944	944	0.3
yn1	20×20	826	884	886	92.780	893.9	27.6	884	891.3	106.3
yn2	20×20	861	907	911	13.070	918.7	27.7	907	911.2	110.4
yn3	20×20	827	892	897	37.220	902.8	43.5	892	895.5	110.8
yn4	20×20	918	968	975	114.080	986.9	45.1	969	972.6	108.7
swv01	20×10	1407	1407	1421	68.250	1463.4	64.6	1412	1423.7	142.1
swv02	20×10	1475	1475	1475	136.940	1505.2	58.6	1475	1480.3	119.7
swv03	20×10	1369	1398	1423	61.800	1437.1	57.9	1398	1417.5	139.1
swv04	20×10	1450	1470	1508	223.360	1543.4	98.6	1470	1483.7	143.9
swv05	20×10	1424	1424	1445	149.940	1486	63.7	1425	1443.8	146.7
swv06	20×15	1591	1678	1722	104.360	1753.7	62.1	1679	1700.1	192.5
swv07	20×15	1446	1600	1634	111.040	1668.4	53.2	1603	1631.3	190.2
swv08	20×15	1640	1756	1804	25.802	1829	66.7	1756	1786.9	190

swv09	20×15	1604	1661	1691	130.510	1722.3	58.8	1661	1689.2	193.8
swv10	20×15	1631	1754	1780	18.050	1818.3	32.0	1754	1783.7	184.6

Table 4. The results of TGA performance on 29 instances due to Lawrence (1984)

			TGA				
Instance	**Size**	**BKS**	Best	$\%DEV_{avg}$	T_{best}(s)	Avg	T_{avg}(s)
la01	10 × 5	666	666	0.00	0.000	666	0.000
la02	10 × 5	655	655	0.00	0.015	655	0.020
la03	10 × 5	597	597	0.00	0.016	597	0.089
la04	10 × 5	590	590	0.00	0.015	590	0.023
la05	10 × 5	593	593	0.00	0.000	593	0.000
la06	15 × 5	926	926	0.00	0.000	926	0.000
la07	15 × 5	890	890	0.00	0.000	890	0.017
la08	15 × 5	863	863	0.00	0.000	863	0.000
la09	15 × 5	951	951	0.00	0.000	951	0.000
la10	15 × 5	958	958	0.00	0.000	958	0.000
la11	20 × 5	1222	1222	0.00	0.000	1222	0.000
la12	20 × 5	1039	1039	0.00	0.000	1039	0.002
la13	20 × 5	1150	1150	0.00	0.000	1150	0.000
la14	20 × 5	1292	1292	0.00	0.000	1292	0.000
la15	20 × 5	1207	1207	0.00	0.016	1207	0.041
la16	10 × 10	945	945	0.00	0.094	945	2.134
la17	10 × 10	784	784	0.00	0.016	784	0.050
la18	10 × 10	848	848	0.00	0.015	848	0.039
la20	10 × 10	902	902	0.00	0.031	902	0.103
la22	15 × 10	927	927	0.00	0.109	927	5.755
la23	15 × 10	1032	1032	0.00	0.047	1032	0.049
la26	20 × 10	1218	1218	0.00	0.078	1218	0.100
la28	20 × 10	1216	1216	0.00	0.109	1216	0.290
la30	20 × 10	1355	1355	0.00	0.093	1355	0.106
la31	30 × 10	1784	1784	0.00	0.000	1784	0.052
la32	30 × 10	1850	1850	0.00	0.047	1850	0.097
la33	30 × 10	1719	1719	0.00	0.031	1719	0.123
la34	30 × 10	1721	1721	0.00	0.156	1721	0.162
la35	30 × 10	1888	1888	0.00	0.046	1888	0.869