Genetic Algorithms for Solving Open Shop Scheduling Problems

Sami Khuri and Sowmya Rao Miryala

Department of Mathematics and Computer Science
San José State University
San José, California 95192, USA
khuri@cs.sjsu.edu
http://www.mathcs.sjsu.edu/faculty/khuri/index.html

Abstract. In this paper we investigate the use of three evolutionary based heuristics to the open shop scheduling problem. The intractability of this problem is a motivation for the pursuit of heuristics that produce approximate solutions. This work introduces three evolutionary based heuristics, namely, a permutation genetic algorithm, a hybrid genetic algorithm and a selfish gene algorithm, and tests their applicability to the open shop scheduling problem. Several problem instances are used with our evolutionary based algorithms. We compare the results and conclude with some observations and suggestions on the use of evolutionary heuristics for scheduling problems. We also report on the success that our hybrid genetic algorithm has had on one of the large benchmark problem instances: our heuristic has produced a better solution than the current best known solution.

1 Introduction

An Open Shop Scheduling Problem (OSSP) involves a collection of m machines and a collection of n jobs. Each job comprises of a collection of operations sometimes called tasks. Each machine can process at most one operation at a time and each job can be processed by at most one machine at any given time. The order in which the jobs are processed on the machine, and the order in which the job is processed by the machines can be chosen arbitrarily. The goal of the open shop scheduling problem is to determine a feasible combination of the machine and job orders, in other words, a schedule, which minimizes the overall finishing time, also known as makespan.

The OSSP has many applications, especially in the manufacturing world and in industry. Consider for instance a large automotive garage with specialized shops. A vehicle may require the following work (job in our terminology): replace exhaust pipe and muffler, align the wheels and tune up. The three operations of that one job may be carried out in any order. However, the exhaust system, alignment, and tune-up shops are in different buildings, and it is therefore not possible to perform two operations simultaneously. We also assume that preemption is not at all desirable. In the OSSP, we assume that we have several such jobs, i.e., several vehicles that need repair.

The OSSP belongs to the class of NP-hard problems. The partition problem can be reduced to it [11]. Thus, unless $P = NP$, the search for exact algorithms is replaced by the design of approximate algorithms. Incidently, only a handful of exact algorithms do exist. A relatively fast branch and bound method for the OSSP, that solved some benchmark problems to optimality, was recently developed [3]. Other methods include one-to-one mapping of feasible schedules to special Latin rectangles [1], and applying insertion techniques with rank minimal schedules [2].

In the next section, we give the formal definition of the open shop scheduling problem. The paper then shifts to the evolutionary based heuristics. The different ingredients with justifications, for our genetic algorithm (GA) implementations, are then presented. More precisely, we discuss the Permutation GA and the Hybrid GA that use LibGA [4]. The third evolutionary based heuristic, the Selfish Gene Algorithm, is then introduced and discussed. In the fourth section, several problem instances are used with the three evolutionary based heuristics and the results are compared. We also report on the problem instance for which our Hybrid GA produced a better solution than the currently best known solution. Our work concludes with some observations from our findings, and some suggestions on the use of evolutionary heuristics for scheduling optimization problems.

2 The Open Shop Scheduling Problem

The open shop scheduling problem consists of m machines $M_1, M_2, \ldots, M_m$ and n jobs $J_1, J_2, \ldots, J_n$. Each job J_i consists of m operations O_{ij} ($j = 1$ to m) where O_{ij} has to be processed on machine M_j for P_{ij} units of time without preemption. Furthermore, as mentioned in the previous section, we assume that each machine can process at most one operation at a time and each job can be processed by at most one machine at any given time. The machine order for each machine: the order in which the jobs are processed on the machine, and the job order for each job: the order in which the job is processed by the machines, can be chosen arbitrarily. The open shop scheduling problem is an optimization problem in which we are to determine a feasible combination of the machine and job orders, that is a schedule, which minimizes a certain objective function: the makespan.

A feasible OSSP schedule assigns a start time to each operation, satisfying the constraint that a machine can only process one operation at a time and that two or more operations from the same job cannot be processed at the same time. The main objective is to generate a schedule with a makespan as short as possible, where the makespan is the total elapsed time in the schedule. A non-preemptive schedule is one in which the operation being processed by a machine cannot be interrupted. In other words, the difference between the finish time and the start time for a particular operation is equal to the length of the operation i.e., finish time - the start time = P_{ij} [11].

Consider the 4×4 benchmark problem depicted in Table 1. Note that the number of jobs is equal to the number of machines. Most benchmark problems in the literature of scheduling have this property. A schedule to the problem instance of Table 1 is given in Table 2. We note that the operations are not scheduled in their order of appearance in Table 1. Thus, operation O_{32}, for instance, is scheduled at time 78 while operation O_{31} is scheduled at time 226, as can be seen in Table 2. Operation O_{22} is the last one to finish, with "end time" equal to 293, thus, the makespan of the schedule given in Table 2 is 293, which happens to be the optimal solution for this problem.

Machines	Job J_1	Job J_2	Job J_3	Job J_4
M_1	85	23	39	55
M_2	85	74	56	78
M_3	3	96	92	11
M_4	67	45	70	75

Table 1. A 4×4 benchmark problem for the OSSP

Machine	Job	Operation	Operation Length	Start time	End Time
M_1	J_1	O_{11}	85	0	85
M_2	J_4	O_{24}	78	0	78
M_3	J_2	O_{32}	96	0	96
M_4	J_3	O_{43}	70	0	70
M_4	J_4	O_{44}	75	78	153
M_2	J_3	O_{23}	56	78	134
M_1	J_2	O_{12}	23	96	119
M_3	J_1	O_{31}	3	96	99
M_3	J_3	O_{33}	92	134	226
M_1	J_4	O_{14}	55	153	208
M_2	J_1	O_{21}	85	134	219
M_4	J_2	O_{42}	45	153	198
M_4	J_1	O_{41}	67	219	286
M_1	J_3	O_{13}	39	226	265
M_2	J_2	O_{22}	74	219	293
M_3	J_4	O_{34}	11	226	237

Table 2. A schedule for the benchmark problem in Table 1 with the makespan = 293

In the next section, we introduce the three evolutionary based heuristics we implemented and ran with OSSP benchmark problems taken from a well known source of test problems.

3 Genetic Algorithms for OSSP

In this work, we use three genetic algorithms. We start our discussion by presenting the Permutation GA and then turn our attention to the Hybrid GA.

We finally introduce a relatively new variant of GA known as the Selfish Gene Algorithm[5]. For each genetic algorithm, we introduce the chromosome representation and the stochastic operators of crossover and mutation. All three genetic algorithms use the makespan as fitness function. In other words, every chromosome in the population is decoded yielding a schedule, and that schedule has a makespan which determines how "fit" that chromosome is.

3.1 The Permutation Genetic Algorithm

Each operation of a job is given a unique number. We first number the operations of job J_1, then the operations of job J_2, and so on. Thus, for a problem with *3* jobs and *3* machines, the first operation of job J_1 will be given number 1, and the last operation of job J_3 will be given number 9. For the individual chromosomes of the GA, we use strings of length p, where p is the total number of operations involved. A scheduling of the operations is represented by using a non-binary string, $y_1, y_2, \ldots, y_p$, where the value of y_i represents the operation to be scheduled next. Thus, if we have *3* jobs and *3* machines, the following string represents a scheduling of the nine operations: The chromosome of Figure 1

1	2	3	4	5	6	7	8	9
7	5	9	3	1	8	6	2	4

Fig. 1. Chromosome representation for the 3×3 OSSP

is interpreted as follows. First schedule the operation that has number 7, then operation with the number 5, and so on.

In our study, we use generational genetic algorithm with roulette wheel selection, uniform crossover, and swap mutation. Since our goal is to *minimize* the makespan, each chromosome in the population occupies a slot on the roulette wheel of size *inversely* proportional to its fitness value. The procedure for calculating makespan is as follows:

procedure makespan
begin
    i = 0;
    while i is less than the number of jobs times the number of machines **do**
    // loop count = length of chromosome
    begin
        determine the job number;
        determine the machine number;
        get the length of the operation;
        schedule the operation;
    end
    return the end time of the machine that finishes last (this is
    the makespan of the current chromosome that we are evaluating);
end.

Procedure **makespan**'s loop starts by determining the job number, machine number and the operation length. The next step consists in scheduling the operations. Our goal is to schedule an operation at the earliest possible time. Scheduling an operation at the earliest possible time is not as obvious as it might seem.

Consider the following scenario (see Figure 2): we want to schedule operation c from job J_c of length t on machine M_j.

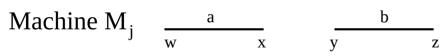

Fig. 2. Scheduling operation c between operations a and b on machine M_j

Our algorithm scans the operations that have been scheduled on M_j. If there is a "gap" between two consecutive operations, a and b such that: $y-x \geq t$, where x is the finishing time of a and y is the starting time of b, then we check to see if operation c could be scheduled between times x and y. Operation c is schedulable between x and y only if no other operation from job J_c is being processed on some other machine between times x and y. If no such gap exists, then operation c is scheduled sometime after the last operation that was processed on machine M_j.

3.2 The Hybrid Genetic Algorithm

Genetic algorithms along with heuristics are used in this approach. Hybridizing a GA with a heuristic to search the space for candidate solutions can be found throughout the literature of genetic algorithms[9].

The chromosome is a string of length p, where p is the total number of operations involved. Each gene can take alleles in the range $1, 2, ..., j$, where j is the largest job number. Thus, if we have *3* jobs and *3* machines, the string of Figure 3 represents a scheduling of the nine operations.

1	2	3	4	5	6	7	8	9
1	2	1	3	1	2	3	1	1

Fig. 3. Chromosome representation for the 3×3 OSSP

The chromosome of Figure 3 is interpreted in the following way. First, we choose an untackled operation for the first job, and schedule it. Then choose an untackled operation from the second uncompleted job. Since the third position of the chromosome in Figure 3 has the value 1, schedule an untackled operation from the first job again.

Building a schedule is accomplished by the schedule builder, which maintains a circular list of uncompleted jobs and a list of untackled operations for each job. Considering the "next uncompleted job" is taken modulo the length of the circular list. For example, by the time we get to the 8^{th} position of the chromosome of Figure 3, all the operations of job 1 would have been tackled. Positions 8 and 9 will choose the last untackled operations of jobs 2 and 3, respectively.

We yet have to establish how the choice among the remaining untackled operations is performed. In our implementation, we let the schedule builder choose the operation with the Largest Processing Time (**LPT**), breaking ties according to an a priori ordering of the operations[8].

Once again, our objective is to minimize makespan, which is used as the fitness function. Procedure **hybrid_makespan** used in this approach differs slightly from procedure **makespan** in Section 3.1, since here we have to incorporate the **LPT** heuristic.

```
procedure hybrid_makespan
begin
        i = 0;
        while i less than the number of jobs times the number of machines do
        // loop count = length of chromosome
        begin
                determine the uncompleted job number;
                determine the machine number for the job that has the
                longest operation processing time;
                get the length of the operation;
                schedule the operation;
        end
        return the end time of the machine that finishes last (this is
        the makespan of the current chromosome that we are evaluating);
end.
```

Once the job number, machine number and the operation length are determined, the scheduling of the chosen operation proceeds in the same fashion as was described in Section 3.1.

In this implementation we use generational genetic algorithm, with roulette wheel selection, uniform crossover and swap mutation.

3.3 The Selfish Gene Algorithm

The field of Evolutionary Computation is based on search and optimization algorithms that were inspired by the biological model of Natural Selection. Several different algorithmic paradigms, among which we find Genetic Algorithms, Genetic Programming and Evolutionary Programming, were proposed after the Darwinian theory. Their underlying common assumption is the existence of a population of individuals that strive for survival and reproduction. Under this assumption, the basic unit of the evolution is the individual, and the goal of the algorithm is to find an individual of maximal fitness[5], i.e., an individual of smallest fitness value, since the OSSP is a minimization problem.

The work of R. Dawkins[6] has put evolution in a different perspective, where the fundamental unit of evolution is the gene, rather than the individual. This view is not in contrast with classical Darwinism, but provides an alternative interpretation, that is formalized by the Selfish Gene Theory. In this theory, individual genes strive for their appearance in the genotype of individuals, whereby individuals themselves are nothing more than vehicles that allow genes to reproduce. In a population, the important aspect is not the fitness of various individuals, since they are mortal, and their good qualities will be lost with their death. Genes, on the other hand, are immortal, in the sense that a given fragment of chromosome can replicate itself to the offspring of an individual, and therefore it survives its death. Genes are selected by evolution on the basis of their ability to reproduce and spread in the population: the population itself can therefore be seen as a pool of genes. Due to the shuffling of genes that takes place during sexual reproduction, good genes are those that give higher reproduction probabilities to the individuals they contribute to build, when combined with the other genes in the same individual.

Following the Selfish Gene Theory, the Selfish Gene Algorithm (FGA) neither relies on crossover nor needs to model a particular population. Instead it works on a Virtual Population (VP), which models the gene pool concept via statistical measures. To avoid confusion, for each gene we will explicitly distinguish between its location in the genome (the locus) and the value appearing at that locus (the allele). Each potential solution is modeled as a genotype, where each locus can be occupied by one of several possible alleles. In the FGA, different alleles fight to be presented in a specific locus. The "goodness" of each allele is represented by its higher frequency in the Virtual Population. The fight is actually performed at the phenotypic level, via a suitably defined fitness function.

As with other Evolutionary Algorithms, an individual in the FGA is represented by its genome. Let g be the number of loci in the genome; each locus L_i ($i = 1 \dots g$) into the genome can be occupied by different gene values, called alleles. The alleles that can occupy locus L_i are denoted with a_{ij} ($j = 1 \dots n_i$) and are collectively represented as a vector $A_i = (a_{i1}, a_{i2}, \dots, a_{in_i})$.

In the VP, due to the number of possible combinations, genomes tend to be unique, but some alleles might be more frequent than others. In the FGA, the goodness of an allele is measured by the frequency with which it appears in the VP. Let p_{ij} be the marginal probability for a_{ij} that conceptually represents the statistical frequency of the allele a_{ij} in locus L_i within the whole VP, regardless of the alleles found in other loci. Marginal probabilities of alleles in A_i for locus L_i are collected in the vector $P_i = (p_{i,1}, \dots, p_{i,n_i})$. The VP can therefore be statistically characterized by all marginal probability vectors $P_i = (P_1, \dots, P_g)$. Note that P is not a matrix, because the number of alleles for each locus can be different [5].

The pseudo code for the main procedure in the FGA is as follows:

```
procedure FGA
begin
        genome B, G1 , G2;
        initialize all probabilities p_ij to 1/(n_i);
        B = select_individual(); (best so far)
        do
        begin
                G1 = select_individual();
                G2 = select_individual();
                if (fitness(G1) < fitness(G2)) then
                begin
                        reward_alleles(G1);
                        penalize_alleles(G2);
                        if (fitness(G1) < fitness(B)) then B = G1 ;
                end
                else
                begin
                        reward_alleles(G2);
                        penalize_alleles(G1);
                        if (fitness(G2) < fitness(B)) then B = G2 ;
                end
        end
        while (steady_state() == FALSE) ;
        find makespan given by B;
        print the schedule and makespan given by B;
        return B;
end
```

An individual is represented by its chromosome. The number of genes in a chromosome is equal to twice the number of jobs (n), times the number of machines (m). The representation is different from the Hybrid and the Permutation Genetic Algorithms which have only $n \times m$ genes in the chromosome. In the chromosome, the odd numbered loci are interpreted as jobs and the even numbered loci are interpreted as operations within a job.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	2	1	3	1	2	3	1	1	1	2	1	3	1	2	3	1	1

Fig. 4. Chromosome representation for the 3×3 OSSP

We use the above representation because we wanted to avoid using a heuristic. Thus, the chromosome completely determines which operation is to be scheduled next. The chromosome in Figure 4 is interpreted as follows: place the 2^{nd} uncompleted operation of the 1^{st} uncompleted job in the earliest place where it will fit in the schedule, then place the 3^{rd} uncompleted operation of the 1^{st}

uncompleted job in the earliest place where it will fit in the schedule, and so on. Although no heuristic is used to determine the next operation to be scheduled, the schedule builder uses primitive data structure operations to keep track of the untackled operations and jobs.

We calculate our objective function or makespan in the same manner as in procedure **makespan** in Section 3.1.

To determine the efficiency of our three heuristics, we use well known benchmark problems to test them.

4 Experimental Results

Our test data consists of 36 benchmark problems that are well known in the literature [10]. Rather than explicitly giving the data, Taillard gives the pseudocode to generate the individual problem instances. We implemented, in C, Taillard's algorithm and executed it to generate the 36 benchmark problems. The generator takes as input time seed, machine seed and the problem size which are specified in [10]. We divided the benchmark problems into three categories. The small size problems are the 4×4 and 5×5 problem instances. The medium size benchmark problems consist of the 7×7 and 10×10 problem instances. The 15×15 and 20×20 problem instances constitute the large size benchmark problems. The current best and theoretical lower bound for each problem instance can be found in [10] and in [7].

In our study we employ a modified LibGA[4] package with elitism, which we found to be superior to other GA variants. The crossover and mutation rates for both algorithms, the Permutation GA and the Hybrid GA, were 0.6 and 0.1, respectively. As for the FGA, the probability of mutation was set to 0.1, and the value for rewarding or penalizing alleles was set to 0.04. The FGA was terminated after 10,000 iterations or if each locus reached a certain value with a probability of 0.95, whichever occurred first. For comparison reasons, we chose 10,000 iterations, since the other GA algorithms run for 500 generations with a pool size of 200 (thus yielding 10,000 computations).

All three genetic algorithms were run 100 times for each benchmark problem. The best result and the mean of the 100 runs are tabulated in Tables 3, 4, and 5. Table 3 reports the results of the three genetic algorithms on the small size benchmarks. As can be seen in Table 3, the Permutation GA and the FGA obtained the current best solutions for all 10 4×4 benchmark problems, and outperformed the Hybrid GA in all 16 problem instances. As for the 5×5 problem instances, the Permutation GA and the FGA either obtained the current best or produced a schedule whose makespan is within 3% of the current best.

Table 4 reports the results of the three genetic algorithms on the medium size benchmarks. All three algorithms fared better with the 7×7 problem instances than with the 10×10 benchmark problems. Although, the three algorithms were not able to obtain the current best solutions, the Permutation GA produced schedules whose makespan is within 5.5% from the current best, the Hybrid GA produced schedules whose makespan is within 3.5% from the current best, and

Problem Instances				Permutation GA		Hybrid GA		FGA	
Name	Size	Current best	Lower Bound	Best	Mean	Best	Mean	Best	Mean
in41.dat	4×4	193	186	**193**	194	213	213	**193**	194
in42.dat	4×4	236	229	**236**	240	240	244	**236**	241
in43.dat	4×4	271	262	**271**	271	293	293	**271**	272
in44.dat	4×4	250	245	**250**	252	253	255	**250**	252
in45.dat	4×4	295	287	**295**	299	303	304	**295**	299
in46.dat	4×4	189	185	**189**	192	209	219	**189**	194
in47.dat	4×4	201	197	**201**	202	203	203	**201**	203
in48.dat	4×4	217	212	**217**	219	224	224	**217**	220
in49.dat	4×4	261	258	**261**	264	281	281	**261**	265
in410.dat	4×4	217	213	**217**	219	230	230	**217**	221
in51.dat	5×5	300	295	301	312	323	324	302	313
in52.dat	5×5	262	255	**262**	271	269	279	**262**	274
in53.dat	5×5	326	321	331	345	353	355	**326**	343
in54.dat	5×5	312	307	**312**	328	327	339	318	328
in55.dat	5×5	353	349	**353**	367	373	376	357	371
in56.dat	5×5	326	321	**326**	340	341	343	**326**	341

Table 3. Test results for 4×4 and 5×5 benchmarks

the FGA produced schedules whose makespan is within 5.5% from the current best.

Table 5 reports the results of the three genetic algorithms on the large size benchmark problems. As can be seen in Table 5, the Hybrid GA outperformed by far the Permutation GA and the FGA in all 10 problem instances. The Permutation GA produced schedules whose makespan is within 6.5% from the current best, and the FGA produced schedules whose makespan is within 9.5% from the current best. As for the Hybrid GA, it produced schedules whose makespan is the current best for all 15×15 problem instances, and all but one 20×20 problem instances. It missed finding the current best for problem instance in201.dat by less than 1%. More importantly, for in204.dat, our Hybrid GA produced a schedule whose makespan is 1207, which is better than the current best of 1209.

5 Conclusion

In this work, three evolutionary based heuristics are used as approximation algorithms with the open shop scheduling problem. Several problem instances are used with our evolutionary based algorithms. The results are compared and some observations, and suggestions on the use of evolutionary heuristics for scheduling problems are given. We also report an improvement on the previously best known solution for a benchmark 20×20 OSSP that our hybrid genetic algorithm produced. It comes as no surprise that the Hybrid GA outperforms the other two genetic algorithms for large problem instances. We observe that although the results obtained in this work are of preliminary nature, we believe

Problem Instances				Permutation GA		Hybrid GA		FGA	
Name	Size	Current best	Lower Bound	Best	Mean	Best	Mean	Best	Mean
in71.dat	7×7	435	435	438	462	447	455	446	462
in72.dat	7×7	449	443	455	477	454	460	463	481
in73.dat	7×7	435	422	443	464	450	456	450	469
in74.dat	7×7	460	458	465	483	467	475	469	485
in75.dat	7×7	400	398	405	426	406	411	411	430
in101.dat	10×10	641	637	667	705	655	672	695	716
in102.dat	10×10	577	577	586	618	581	589	611	624
in103.dat	10×10	538	538	555	583	541	549	611	624
in104.dat	10×10	595	595	627	646	598	618	646	660
in105.dat	10×10	602	596	623	645	605	618	638	655

Table 4. Test results for 7×7 and 10×10 benchmarks

Problem Instances				Permutation GA		Hybrid GA		FGA	
Name	Size	Current best	Lower Bound	Best	Mean	Best	Mean	Best	Mean
in151.dat	15×15	937	937	967	998	**937**	948	990	1016
in152.dat	15×15	871	871	904	946	**871**	886	950	971
in153.dat	15×15	934	934	969	992	**934**	944	992	1012
in154.dat	15×15	893	893	928	962	**893**	905	955	977
in201.dat	20×20	1155	1155	1230	1269	1165	1190	1264	1292
in202.dat	20×20	1257	1257	1292	1346	**1257**	1267	1336	1378
in203.dat	20×20	1256	1256	1315	1353	**1256**	1267	1359	1386
in204.dat	20×20	1209	1204	1266	1305	**1207**	1224	1312	1386
in205.dat	20×20	1289	1289	1339	1380	**1289**	1293	1390	1417
in206.dat	20×20	1241	1241	1307	1343	**1241**	1246	1342	1381

Table 5. Test results for 15×15 and 20×20 benchmarks

that hybrid methods will have a lot to offer in the future. In our work, we reject the conventional wisdom of using binary representations in favor of more direct encodings. Our non-binary chromosomes represent implicit instructions for building a schedule. One last advantage genetic algorithms has over other more traditional methods is that genetic algorithms are far easier to parallelize than typical knowledge-based and operations research methods.

References

1. Brasel, H., Kleinau, M.: On the Number of Feasible Schedules of the Open Shop Problem - An Application of Special Latin Rectangles: Optimization, **23** (1992) 251–260
2. Brasel, H., Tautenhan, T., Werner, F.: Constructive Heuristic Algorithms for the Open Shop Problem: Computing, **51** (1993) 95–110
3. Brucker, P., Hurink, J., Jirisch, B., Wostmann, B.: Branch and Bound Algorithm for the Open Shop Problem: Discrete Applied Mathematics, **76** (1997) 43–59

4. Corcoran A., Wainwright, R.: LibGA: A User-Friendly Workbench for Order-Based Genetic Algorithm Research. Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, ACM Press, (1993) 111–117
5. Corno, F., Reorda, M., Squillero, G.: The Selfish Gene Algorithm: A New Evolutionary Optimization Strategy. ACM/SAC. (1998)
6. Dawkins, R.: The Selfish Gene. Oxford University Press (1989)
7. Fang, H., Ross, P., Corne, D.: A Promising Hybrid GA/Heuristic Approach for Open Shop Scheduling Problems. DAI Research Paper No. 699, Proceedings of the 11^{th} European Conference on Artificial Intelligence. John Wiley and Sons (1994) 590–594
8. Fang, H., Ross, P., Corne, D.: A Promising Genetic Algorithm Approach to Job Shop Scheduling, Rescheduling and Open Shop Scheduling Problems. DAI Research Paper No. 623, Proceedings of the Fifth International Conference on Genetic Algorithms. (1993) 375–382
9. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolutionary Programs. Springer Verlag. 3rd edition (1996)
10. Taillard, E.: Benchmarks for Basic Scheduling Problems. European Journal of Operations Research. **64** (1993) 278–285
11. Teofilo, G., Sahni, S.: Open Shop Scheduling to Minimize Finish Time. Journal of the Association for Computing Machinery. **23(4)** (1976) 665–679