Job Shop Scheduling using the Clonal Selection Principle

Carlos A. Coello Coello[1], Daniel Cortés Rivera[2] and Nareli Cruz Cortés[3]
CINVESTAV-IPN (Evolutionary Computation Group)
Depto. de Ingeniería Eléctrica, Sección de Computación
Av. IPN No. 2508, Col. San Pedro Zacatenco
México, D. F. 07300, MEXICO
[1]ccoello@cs.cinvestav.mx
[2]dcortes@computacion.cs.cinvestav.mx
[3]nareli@computacion.cs.cinvestav.mx

Abstract

In this paper, we propose an algorithm based on an artificial immune system to solve job shop scheduling problems. The approach uses clonal selection, hypermutations and a mechanism that explores the vicinity of a reference solution. It also uses a decoding strategy based on a search that tries to eliminate gaps in a schedule as to improve the solutions found so far. The proposed approach is compared with respect to three other heuristics using a standard benchmark available in the specialized literature. The results indicate that the proposed approach is very competitive with respect to the others against which it was compared. Our approach not only improves the overall results obtained by the other heuristics, but it also significantly reduces the CPU time required by at least one of them.

Introduction

The purpose of scheduling is to allocate a set of (limited) resources to tasks over time [1]. Scheduling has been a very active research area during several years, both in the operations research and in the computer science literature [2,3] with applications in several disciplines. Research on scheduling basically focuses on finding ways of assigning tasks (or jobs) to machines (i.e., the resources) such that certain criteria are met and certain objective (or objectives) function is optimized.

A wide variety of scheduling problems (e.g., job shop, flowshop, production, etc.) have been tackled with diverse heuristics such as evolutionary algorithms [3,4,5], tabu search [6], and simulated annealing [7], among others. Note, however, that the use of artificial immune systems for the solution of scheduling problems of any type has been scarce (see for example [8,9]).

This paper extends our previous proposal of a new approach based on an artificial immune system (basically on the clonal selection principle) to solve job scheduling problems, which was introduced in [10]. Three are the main changes with respect to our previous proposal are the following: (1) we no longer use a library of antibodies, (2) we introduced two new domain-specific mutation operators, and (3) we use a new backtracking mechanism. As we will see later on, these changes introduce important improvements in our algorithm with respect to the original version. The proposed approach is compared with respect to GRASP (Greedy Randomized Adaptive Search Procedure), a Hybrid Genetic Algorithm (in which

local search is used), a Parallel Genetic Algorithm and our previous AIS [10] in several test problems taken from the specialized literature. Our results indicate that the proposed approach is a viable alternative for solving efficiently job shop scheduling problems and it also improves on our previous version reported in [10].

Statement of the Problem

In this paper, we will be dealing with the Job Shop Scheduling Problem (JSSP), in which the general objective is to minimize the time taken to finish the last job available (makespan). In other words, the goal is to find a schedule that has the minimum duration required to complete all the jobs [2]. More formally, we can say that in the JSSP, we have a set of n jobs $\{J_j\}_{1\leq j\leq n}$, that have to be processed by a set of m machines $\{M_r\}_{1\leq r\leq m}$. Each job has a sequence that depends on the existing precedence constraints. The processing of a job J_j in a machine M_r is called operation O_{jr}. The operation O_{jr} requires the exclusive use of M_r for an uninterrupted period of time p_{jr} (this is the processing time). A schedule is then a set of duration times for each operation $\{c_{jr}\}_{1\leq j\leq n,1\leq r\leq m}$ that satisfies the previously indicated conditions. The total duration time required to complete all the jobs (makespan) will be called L. The goal is then to minimize L.

Garey and Johnson [11] showed that the JSSP is an **NP-hard** problem and within its class it is indeed one of the least tractable problems [3]. Several enumerative algorithms based on *Branch & Bound* have been applied to JSSP. However, due to the high computational cost of these enumerative algorithms, some approximation approaches have also been developed. The most popular practical algorithm to date is the one based on *priority rules* and *active schedule generation* [12]. However, other algorithms, such as an approach called *shifting bottleneck* (SB) have been found to be very effective in practice [13]. The only other attempt to solve the JSSP using an artificial immune system that we have found in the literature is the proposal presented in [8,9] and our previous version of the algorithm presented here [10] (whose differences with our current proposal have been previously indicated). In [8,9], the authors use an artificial immune system in which an antibody indirectly represents a schedule, and an antigen describes a set of expected arrival dates for each job in the shop. The schedules are considered to be dynamic in the sense that sudden changes in the environment require the generation of new schedules. The proposed approach compared favorably with respect to a genetic algorithm using problems taken from [14]. However, the authors do not provide the problems used nor their results.

Description of our Approach

As indicated in [17], an artificial immune system is an adaptive system, inspired on our immune system (its observed functions, principles and models), and intended to be used as a problem-solving tool. Our approach is based on the clonal selection

principle, and can be seen as a variation of an specific artificial immune system called CLONALG, which is has been successfully used for optimization [15]. CLONALG uses two populations: one of antigens and another one of antibodies. When used for optimization, the main idea of CLONALG is to reproduce individuals with a high affinity, then apply mutation (or blind variation) and select the improved maturated progenies produced. Note that "affinity" in this case, is defined in terms of better objective function values rather than in terms of genotypic similarities (as, for example, in pattern recognition tasks), and the number of clones is the same for each antibody. This implies that CLONALG does not really use antigens when solving optimization problems, but, instead, the closeness of each antibody to the global optimum (measured in relative terms with respect to the set of solutions produced so far) defines the rate of hypermutation to be used. It should also be noted that CLONALG does not use a mechanism that allows a change of the reference solution as done with the approach reported in this paper. In order to apply an artificial immune system to the JSSP, it is necessary to use a special representation. In our case, each individual represents the sequence of jobs processed by each of the machines. An antibody is then a string with the job sequence processed by each of the machines (of length $m \times n$). An antigen is represented in the same way as an antibody. The representation adopted in this work is the so-called *permutations with repetitions* proposed in [16] (see an example in Table 1).

Job	**machine(time)**			
1	1(2)	2(2)	3(2)	4(2)
2	4(2)	3(2)	2(2)	1(2)
3	2(2)	1(2)	4(2)	3(2)
4	3(2)	4(2)	1(2)	2(2)
5	1(2)	2(2)	3(4)	4(1)
6	4(3)	2(3)	1(1)	3(1)

Table 1: A problem of size 6 x 4

Input data include the information regarding the machine in which each job must be processed and the duration of this job in each machine. Gantt diagrams are a convenient tool to visualize the solutions obtained for a JSSP. An example of a Gantt diagram representing a solution to the $6 \text{ x } 4$ problem previously indicated is shown in Step 1 of Figure 1 also requires some further explanation:

- The string at the bottom of Figure 1 corresponds to the solution that we are going to decode.
- **Step 1**: This shows the decoding before reaching the second operation of job 2.
- **Step 2**: This shows the way in which job 2 would be placed if a normal decoding was adopted. Note that job 2 (J_2) is shown to the extreme right of machine 3 (M_3).
- **Step 3**: Our approach performs a local search to try to find gaps in the current schedule. Such gaps should comply with the precedence

constraints imposed by the problem. In this case, the figure shows job 2 placed on one of these gaps for machine 3.

- **Step 4**: In this case, we apply the same local search procedure (i.e., finding available gaps) for the other machines. This step shows the optimum solution for this scheduling problem.

Our approach extends the algorithm (based on clonal selection theory) proposed in [17] using a local search mechanism that consists of placing jobs in each of the machines using the available time slots.

Requiere: Input file (in the format adopted in [18]).
Input parameters: *#antigens*, *mutation rate*, *random seed* (optional), *degree of freedom*

p - number of iterations

i - counter

Retrieval of problem (read file) and algorithm's parameters.
Generate (randomly) an *antigen* (i.e., a sequence of jobs) and decode it.
Generate (randomly) an *antibody*.

repeat
  Decode the *antibody*.
  if (the (*antibody - degree of freedom*) is better than the *antigen1*) ***then***
    Make the *antigen1* the same as the *antibody*
    if (the *antibody* is better than the *antigen2*) ***then***
      Make the *antigen2* the same as the *antibody*
    end if
  end if
  Generate a *clone* of the *antibody*
  Mutate the *clone* generated
  Select the best *antibody*
until $i > p$
Report the best solution found, stored in *antigen2*

Algorithm 1: Our AIS for job shop scheduling

Our approach is described in Algorithm 1. First, we generate the initial population. What we do is to randomly generate an antibody and an antigen (it is important to keep in mind that we use a special representation and that both the antibody and the antigen have the same structure). To generate these two elements (antibody and antigen), we adopt a string of length $m \times n$, which is filled with m values ranging from 0 to $n-1$. Once we fill in the array, we perform a set of random permutations in order to obtain the individual to start the search. At the next stage, the main cycle of the algorithm is executed. Within this cycle, we first decode the antibody (note that the antigen was decoded at a previous step).

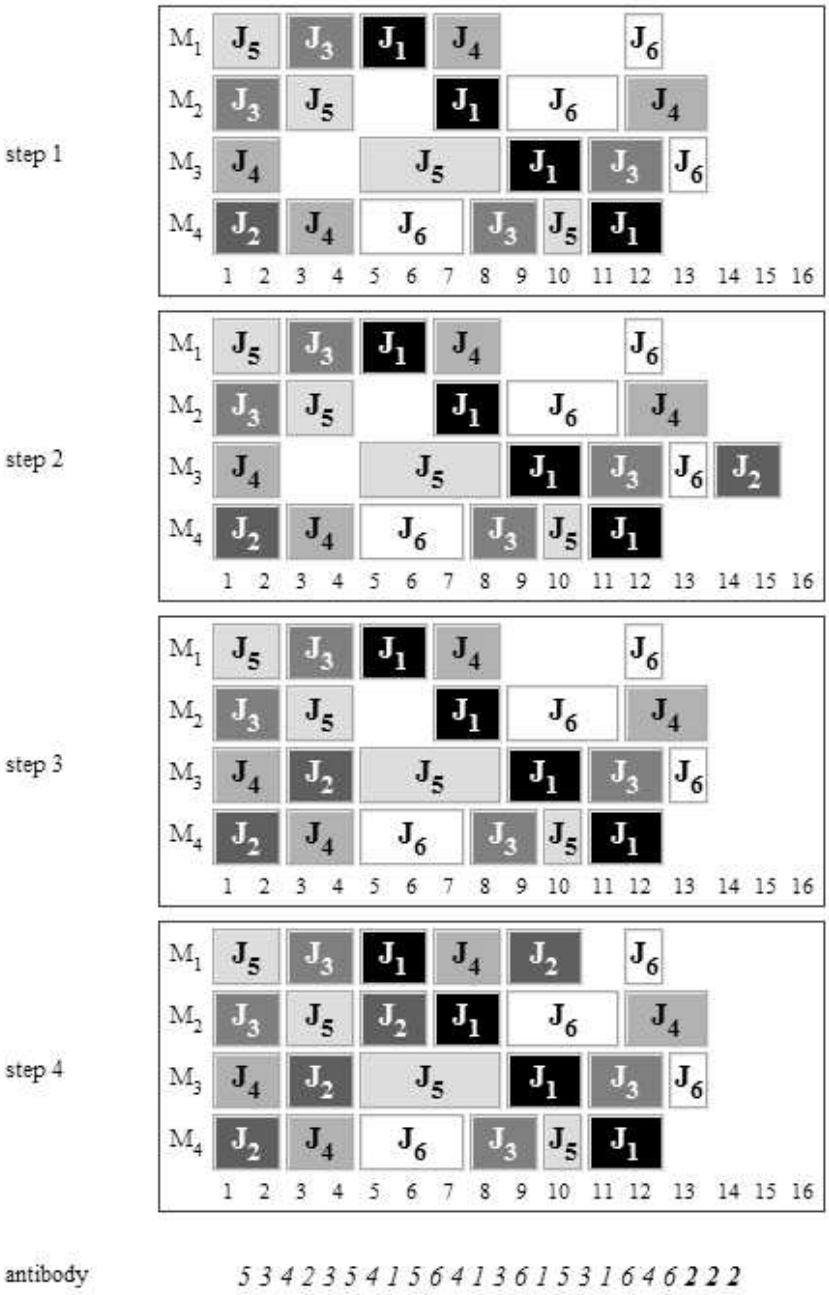

Figure 1: The graphical representation of a solution to the 6×4 problem shown in Table 1 using a Gantt diagram. The string at the bottom of the figure indicates the antibody that we are going to decode. See the text for an explanation of the different steps included.

The process required to decode an individual as to determine its fitness (i.e., the makespan of the corresponding schedule) is the following:

1. We need to have in a matrix all the problem's data (i.e., the processing order of each of the jobs to be handled by the machines available as well as their processing times).
2. We read the string encoding a solution in order to identify each of the jobs contained within (each job is represented by a number between 0 and *m*).
3. Once we know the corresponding job number, we keep a count of the order of occurrence of each of the numbers as to identify the corresponding job operations (i.e., if this is the first occurrence, then it corresponds to the first job operation). We also determine the machine in which each job is processed using the corresponding input matrix.
4. The following step is to place the operation in the schedule. In order to do this, we use a structure that has been previously initialized and which contains the schedule with the necessary information to accommodate the operations without violating any of the constraints of the problem.
5. In order to place an operation in its corresponding place in the schedule, we provide an example in Figure 1. In this figure we can see that the operation is first placed in its corresponding machine (based on the input matrix). After that, we try to locate a gap in the schedule in which we can place this operation, avoiding to interfere with other operations and avoiding to violate the existing constraints.
6. This process of finding gaps to place operations may cause that several strings encoding different orderings can be decoded to the same solution.
7. The next step is to reorder the string encoding a solution such that the next time that such string is decoded it becomes unnecessary to apply the strategy previously described to find gaps. The ordering performed is based on the order of appearance of each operation and considering each machine from the first to the last. By adopting these criteria, we minimize the amount of possible gaps available in the next iteration.
8. Once we have finished this ordering, we create a data structure that is very important for the mutation operator. Such a data structure consists of generating an ordering of the operations per machine such that it is easy to know the position of each operation and the machine to which it belongs without having to check this in an exhaustive manner.
9. We report the corresponding makespan.

The decoding process is the most expensive (computationally speaking) part of our algorithm.

In the next stage of the algorithm, we compare the antigen with respect to the antibody. Note that we do not adopt a phenotypic similarity metric as the affinity measure. Instead, we use the makespan value as our affinity measure. During this process, we use the best solution found so far as a reference for further search (this is called *antigen1* in Algorithm 1). Each time a better solution is found, it is used as a new reference. In the original version of our algorithm [10], we used a single antigen as a reference. However, we decided to keep a second antigen to allow

good (but not the best) solutions to be used as references as well (this is called *antigen2* in Algorithm 1). What we do is to keep a second solution that is one or two units away (in terms of makespan value) from the best solution found so far and we also use it as a reference. This second antigen serves as some sort of backtracking mechanism of the algorithm that allows it to escape from local optima. By using this second antigen, we were able to obtain significant gains in terms of computational time. Once we finish the verification stage in which we check if any of our two antigens (or both) have been improved, the following stage is the cloning of the antibody. This cloning stage consists of copying the antibody a certain number of times without doing any changes to its structure. The number of clones to be produced was varied from 1 to 10 depending on the complexity of the problem tackled. Note however, that if many clones are adopted, the improvement gained is only marginal and the high computational cost increase makes this option unattractive. Thus, we adopted values of either 1 or close to 1 for the number of clones to be produced. Once the clones are available, each of them is mutated in such a way that they suffer a slight variation in in their structure. The algorithm has two types of mutation operators available, and the one to be used is selected with a 50% probability. The similarities and differences between these two mutation operators (which we will call Mutation-A and Mutation-B) are the following:

- In both cases, the mutation operator is applied by using $flip(pm)$ at each string location.[1]
- In both cases, the mutation rate is a function of the antibodies length and it is defined such that 1 mutation takes place for each string (i.e., antibody).
- In both cases, the operator locates an operation, then finds another operation of another job and then swaps the positions of the 2 operations.
- In order to have a quick indexing of the positions of each operation, we use the data structure previously created for the current antibody.
- The only difference between the two mutation operators is that in the case of Mutation-A, we find the first operation and then locate the other operation with which it will swap places. However, if there are other operations of the same job before the current operation, we traverse them all. As a consequence, we not only change those operations, but we also produce more changes to the schedule.
- In the case of Mutation-B, we only locate two operations and swap their locations without any further exploration.

These are all the processes performed by our algorithm. Once it reaches its convergence criterion (a maximum number of iterations), the algorithm reports the best solution found during the process, which is stored in one of the two reference antigens (in *antigen2*). The algorithm then reports the full schedule with all the detailed information regarding the ordering of the machines and the initial and termination times for each of the available operations.

---

[1] The function $flip(pm)$ returns TRUE with a probability pm.

Comparison of Results

We compare our Artificial Immune System (AIS) with respect to 3 different approaches: a Hybrid Genetic Algorithm (HGA) reported in [19], a GRASP approach [20], and a Parallel Genetic Algorithm (PGA) [21]. We chose these references for two main reasons: (1) they provide enough information (e.g., numerical results) as to allow a comparison; (2) these algorithms have been found to be very powerful in the job shop scheduling problem studied in this paper. Note that the test problems adopted were taken from the OR-Library [18]. Additionally, we also compared results with respect to our previous AIS [10]. All our tests were performed on a PC with an Intel Pentium 4 running at 2.6 GHz with 512 MB of RAM and using Red Hat Linux 9.0. Our approach was implemented in C++ and was compiled using the GNU g++ compiler.

	deviation	**Deviation AIS**	**Improvement**
HGA	0.42%	0.18%	0.23%
GRASP	0.47%	0.18%	0.28%
PGA	0.93%	0.18%	0.74%

Table 2: Comparison of results between our Artificial Immune System (AIS) and three other algorithms: Greedy Randomized Adaptive Search Procedure (GRASP) [20], the Hybrid Genetic Algorithm (HGA) [19], and the Parallel Genetic Algorithm (GA) [21].

Table 2 shows the overall comparison of results. In the first column, we show the algorithm with respect to which we are comparing our results. In the second column, we show the average deviation of the best results obtained by each algorithm with respect to the best known solution for the 43 test problems adopted in our study. In the third column, we show the average deviation of our AIS with respect to the best known solution for the 43 test problems adopted in our study. The last column indicates the improvement achieved by our AIS with respect to each of the other algorithms compared. From Table 2, we can see that our approach was able to improve on the overall results produced by the 3 other techniques. The most remarkable improvement produced was with respect to the PGA [21].

	AIS		
	Win	**Tie**	**Lose**
HGA	3	32	8
GRASP	3	30	10
PGA	0	23	17

Table 3: Overall performance of our AIS with respect to the 3 other algorithms against which it was compared. The column labeled **Win** shows the number of problems in which each algorithm beat our AIS. The column labeled **Tie** indicates ties between our AIS and the other algorithms. Finally, the column labeled **Lose** indicates the number of problems in which each algorithm lost with respect to our AIS.

In Table 3, we show the overall performance of our AIS with respect to the 3 other algorithms against which it was compared. Results indicate that the HGA beat our

AIS in 3 problems and it lost in 8. In the remainder (32 problems), they tied. GRASP beat our AIS in 3 problems and lost in 10. The worst contender was the PGA, which was not able to beat our AIS in any problem and lost in 17 problems.

Table 4 summarizes the results obtained by each of the 4 approaches compared in the 43 test problems taken from the OR-Library [18]. We use **boldface** to indicate both the best known results and when an algorithm reached such result. Note that the number of evaluations performed is only reported for our two AIS and for GRASP. The reason is that we only found such information available for GRASP. We can clearly see that our AIS obtained competitive results with respect to the other approaches compared. Furthermore, the number of evaluations performed by our AIS was significant lower than those performed by GRASP.[2] Some remarkable examples are the following:

- **FT10**: In this problem, GRASP found the best known solution, but it required 2.5 million evaluations. Our AIS found a solution which is only 1% away from the best known solution and it only required 250,000 evaluations. Note that in our original AIS produced a poorer solution than the new version, and it required 20 million evaluations.
- **LA28**: In this problem, the best solution found by GRASP is slightly worse than the best solution found by our AIS (1225 vs. 1216). However, our AIS required 1 million evaluations whereas GRASP required 20 million iterations. Note that the original version of our AIS found a worse solution using 5 million evaluations.
- **LA16**: Both GRASP and our AIS reached the best known solution. However, GRASP required 1.3 million evaluations and our approach required only 10,000 evaluations. Note that the original version of our AIS required 2 million evaluations to reach this solution.

As can be seen in Table 4, the new version of the algorithm presents a significant improvement with respect to the original version reported in [10], both in terms of the quality of the solutions obtained as in terms of the computational efforts required to obtain them.

Instance	Size (m x n)	BKS	HGA	AIS	Evals AIS	oAIS	Evals oAIS	GRASP	Iters. GRASP	PGA
FT06	6 x 6	**55**	**55**	**55**	0.0001	**55**	0.001	**55**	0.00001	**55**
FT10	10 x 10	**930**	**930**	936	0.25	941	20	**930**	2.5	936
FT20	20 x 5	**1165**	**1165**	**1165**	0.5	-	-	**1165**	4.5	1177
LA01	10 x 5	**666**	**666**	**666**	0.001	**666**	0.01	**666**	0.0001	**666**
LA02	10 x 5	**655**	**655**	**655**	0.01	**655**	0.01	**655**	0.004	666

[2] In fact, what we report as the number of evaluations for GRASP is actually the number of iterations performed by the algorithm. Since at each iteration, GRASP performs several evaluations of the objective function, the real number of evaluations is much higher than those reported in Table 4.

LA03	10 x 5	**597**	**597**	**597**	0.01	**597**	10	**597**	0.01	**597**
LA04	10 x 5	**590**	**590**	**590**	0.001	**590**	0.01	**590**	0.001	**590**
LA05	10 x 5	**593**	**593**	**593**	0.001	**593**	0.01	**593**	0.0001	**593**
LA06	15 x 5	**926**	**926**	**926**	0.001	**926**	0.01	**926**	0.0001	**926**
LA07	15 x 5	**890**	**890**	**890**	0.001	**890**	0.01	**890**	0.0001	**890**
LA08	15 x 5	**863**	**863**	**863**	0.001	**863**	0.01	**863**	0.0003	**863**
LA09	15 x 5	**951**	**951**	**951**	0.001	**951**	0.01	**951**	0.0001	**951**
LA10	15 x 5	**958**	**958**	**958**	0.001	**958**	2	**958**	0.0001	**958**
LA11	20 x 5	**1222**	**1222**	**1222**	0.001	-	-	**1222**	0.0001	**1222**
LA12	20 x 5	**1039**	**1039**	**1039**	0.001	-	-	**1039**	0.0001	**1039**
LA13	20 x 5	**1150**	**1150**	**1150**	0.001	-	-	**1150**	0.0001	**1150**
LA14	20 x 5	**1292**	**1292**	**1292**	0.001	-	-	**1292**	0.0001	**1292**
LA15	20 x 5	**1207**	**1207**	**1207**	0.001	-	-	**1207**	0.0002	**1207**
LA16	10 x 10	**945**	**945**	**945**	0.01	**945**	2	**945**	1.3	977
LA17	10 x 10	**784**	**784**	**784**	0.01	785	2	**784**	0.02	787
LA18	10 x 10	**848**	**848**	**848**	0.01	**848**	2	**848**	0.05	**848**
LA19	10 x 10	**842**	**842**	**842**	0.01	848	10	**842**	0.02	857
LA20	10 x 10	**902**	907	907	0.25	907	5	**902**	17	910
LA21	15 x 10	**1046**	**1046**	**1046**	0.25	-	-	1057	100	1047
LA22	15 x 10	**927**	935	**927**	0.25	-	-	**927**	26	936
LA23	15 x 10	**1032**	**1032**	**1032**	0.25	-	-	**1032**	0.01	**1032**
LA24	15 x 10	**935**	953	**935**	0.25	-	-	954	125	955
LA25	15 x 10	**977**	986	979	0.25	1022	5	984	32	1004
LA26	20 x 10	**1218**	**1218**	**1218**	0.2	-	-	**1218**	3.5	**1218**
LA27	20 x 10	**1235**	1256	1240	0.5	-	-	1269	10.5	1260
LA28	20 x 10	**1216**	1232	**1216**	1	1277	5	1225	20	1241
LA29	20 x 10	**1157**	1196	1170	5	1248	6.4	1203	50	1190
LA30	20 x 10	**1355**	**1355**	**1355**	0.1	-	-	**1355**	3	1356
LA31	30 x 10	**1784**	**1784**	**1784**	0.005	-	-	**1784**	0.01	**1784**
LA32	30 x 10	**1850**	**1850**	**1850**	0.025	-	-	**1850**	0.0001	**1850**
LA33	30 x 10	**1719**	**1719**	**1719**	0.025	-	-	**1719**	0.001	**1719**
LA34	30 x 10	**1721**	**1721**	**1721**	0.01	-	-	**1721**	0.05	1730
LA35	30 x 10	**1888**	**1888**	**1888**	0.05	1903	5	**1888**	0.01	**1888**
LA36	15 x 15	**1268**	1279	1281	0.5	1323	6.4	1287	51	1305
LA37	15 x 15	**1397**	1408	1408	0.5	-	-	1410	20	1441
LA38	15 x 15	**1196**	1219	1204	0.5	1274	6.4	1218	20	1248
LA39	15 x 15	**1233**	1246	1249	0.5	1270	6.4	1248	6	1264
LA40	15 x 15	**1222**	1241	1228	2.5	1258	6.4	1244	2	1252

Table 4: Comparison of results between our artificial immune system (AIS), GRASP (Greedy Randomized Adaptive Search Procedure) [20], HGA (Hybrid Genetic Algorithm) [19], and PGA (Parallel Genetic Algorithm) [21]. oAIS refers to our original AIS, reported in [10] and is included only to have a rough idea of the improvements achieved with the new

version. The number of evaluations reported is in millions. Only the number of evaluations of GRASP and our two AIS versions are reported because this value was not available for the other approaches. We show in **boldface** both the best known solution and the cases in which an algorithm reached such value.

Conclusions and Future Work

We have introduced a new approach based on an artificial immune system to solve job shop scheduling problems. The approach uses concepts from clonal selection theory (extending ideas from CLONALG [15]), and adopts a permutation representation that allows repetitions. The comparison of results indicated that the proposed approach is highly competitive with respect to other heuristics, even improving on their results in some cases. It also improves in the previous version of the algorithm reported in [10]. In terms of computational efficiency, our approach performs a number of evaluations that is considerably lower than those performed by GRASP [21] while producing similar results.

As part of our future work, we intend to add a mechanism that avoids the generation of duplicates (something that we do not have in the current version of our algorithm). It is also desirable to find a set of parameters that can be fixed for a larger family of problems as to eliminate the empirical fine-tuning that we currently perform. Finally, we also plan to work on a multiobjective version of job shop scheduling in which 3 objectives would be considered [3]: 1) makespan, 2) mean flowtime and 3) mean tardiness. This would allow us to generate trade-offs that the user could evaluate in order to decide what solution to choose.

Acknowledgments

The first author acknowledges support from CONACyT project No. 34201-A. The second and third authors acknowledge support from CONACyT through a scholarship to pursue graduate studies in Computer Science at the Sección de Computación of the Electrical Engineering Department at CINVESTAV-IPN.

References

1. M. Pinedo. *Scheduling---Theory, Algorithms, and Systems*. Prentice Hall, Englewood Cliffs, 1995.
2. Kenneth R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, New York, 1974.
3. Tapan P. Bagchi. *Multiobjective Scheduling by Genetic Algorithms.* Kluwer Academic Publishers, New York, September 1999.
4. R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms: I. Representation. *Computers and Industrial Engineering*, 30:983--997, 1996.
5. R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms: II. Hybrid genetic search strategies. *Computers and Industrial Engineering*, 36(2):343--364, 1999.

6. J.W. Barnes and J.B. Chambers. Solving the Job Shop Scheduling Problem using Taboo Search. *IIE Transactions*, 27(2):257--263, 1995.
7. Olivier Catoni. Solving Scheduling Problems by Simulated Annealing. *SIAM Journal on Control and Optimization*, 36(5):1539--1575, September 1998.
8. Emma Hart, Peter Ross, and J. Nelson. Producing robust schedules via an artificial immune system. In *Proceedings of ICEC'98*, pp. 464--469, Anchorage, Alaska, 1998. IEEE Press.
9. Emma Hart and Peter Ross. The Evolution and Analysis of a Potential Antibody Library for Use in Job-Shop Scheduling. In David Corne et al., eds, *New Ideas in Optimization*, pp. 185—202, McGraw-Hill, 1999.
10. Carlos A. Coello Coello, Daniel Cortés Rivera, and Nareli Cruz Cortés. Use of an Artificial Immune System for Job Shop Scheduling. In Jon Timmis et al., eds, *Proceedings of ICARIS'2003*, pp. 1--10, September 2003. Springer-Verlag. Lecture Notes in Computer Science Vol. 2787.
11. David S. Johnson Michael R. Garey. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.
12. Albert Jones and Luis C. Rabelo. *Survey of Job Shop Scheduling Techniques*. National Institute of Standards and Technology, 1998.
13. J. Adams E. Balas and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391-401, 1988.
14. Thomas E. Morton and David W. Pentico. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management.* John Wiley & Sons, 1993.
15. Leandro Nunes de Castro and Fernando José Von Zuben. Learning and Optimization Using the Clonal Selection Principle. *IEEE Transactions on Evolutionary Computation*, 6(3):239--251, 2002.
16. Takeshi Yamada and Ryohei Nakano. Job-shop scheduling. In A.M.S. Zalzala and P.J. Fleming, editors, *Genetic Algorithms in Engineering Systems*, pp. 134--160. The Institution of Electrical Engineers, 1997.
17. Leandro Nunes de Castro and Jonathan Timmis. *Artificial Immune System: A New Computational Intelligence Approach*. Springer Verlag, Great Britain, September 2002. ISBN 1-8523-594-7.
18. J. E. Beasley. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operations Research Society*, 41(11):1069--1072, 1990.
19. José Fernando Goncalves, Jorge José Mendes, and Mauricio G.C. Resende. *A Hybrid Genetic Algorithm for the Job Shop Scheduling Problem*. Technical Report TD-5EAL6J, AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932 USA, September 2002.
20. Renata M. Aiex, S. Binato, and Mauricio G.C. Resende. Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing*, 29(4):393--430, 2003.
21. José Fernando Goncalves and N.C.Beirao. Um algoritmo genético baseado em chaves aleatórias para sequenciamiento de operacoes. *Revista Associacão Portuguesa de Desenvolvimento e Investigacão Operacional*, 19:123 -- 137, 1999. (in Portuguese).