Comparison of a Job-Shop Scheduler using Genetic Algorithms with a SLACK based Scheduler

Nishant Deshpande
Department of Computer Science
Stanford, CA 94305
nishantd@cs.stanford.edu (650) 248 5159
4 June 2002

Abstract: This paper describes an implementation of a scheduler using Genetic Algorithms. It compares the performance of the GA-scheduler with a scheduler based on a conventional search based on the SLACK heuristic. Optimal parameters for the GA-scheduler are empirically found and discussed for theoretical compatibility. The GA-scheduler gives good results but is not competitive with the SLACK based conventional scheduler.

1. Introduction

Genetic Algorithms offer a 'one size fits all' solution to problem solving involving search. Unlike other conventional search alternatives, GA's can be applied to most problems out of the box, only needing a good function specification to optimize and a good choice of representation and interpretation. This, coupled with the exponentially increasing speed/cost ratio of computers, make them a choice to consider for any search problem.

Scheduling is a very important field for many practical and theoretical reasons. Most industries have logistical or scheduling problems which they would like to optimise. One of the problems that serves as a model for the general problem is 'job-shop scheduling'.

The job shop scheduling problem originates from the manufacturing domain. The basic problem is that of scheduling N jobs, given each job has one or more sequencing constraints with other jobs, the (perhaps exclusive) use of a resource, a processing time, and associated ready time (the earliest time the job can start) and deadline (the latest time the job can finish). The goal is to minimize the total time taken (the time-span) while meeting all of the above constraints.

In the general case, the scheduling problem is NP-complete. This has lead to considerable research in heuristic procedures to solve the problem. Several different approaches have been tried. Most frequently, the problem has been formulated as one of finding a consistent assignment of start times for each job. Recently, the problem has been formulated as that of assigning sufficient ordering constraints between jobs vying for the same resource. This represents a set of feasible solutions (the set of feasible start times for all jobs). A heuristic, SLACK, is used in the search (described in Section 2).

The above mentioned strengths of Genetic Algorithms come with a price. GA's do not exploit domain information in the search. They rely on good representation and fast scoring function evaluation.

Section 2 briefly describes the job shop scheduling problems addressed in this paper and the conventional SLACK-based scheduler. This will help in the understanding of representation issues later in the paper. Section 3 describes the possible representations of a schedule in a GA, and discusses strengths and weaknesses of each. Section 4 presents the tableau for the GA used, and the scoring function used. Section 5 presents the experiment setup, results and discussion of the results. Section 6 concludes and Section 7 details possible future work.

2. Sadeh job-shop benchmarks and SLACK-based scheduling

2.1 Sadeh job-shop benchmarks

Norman Sadeh designed a suite of job-shop problems that have been used as benchmarks. A typical problem (shortened) is illustrated below.

```
length(0) = 14          // job 0 is defined and has length 14
needs(0,0)              // job 0 needs resource 0
length(1) = 21          // job 1 is defined and has length 21
needs(1,0)              // job 1 needs resource 1
length(2) = 7           // job 2 is defined and has length 7
needs(2,1)              // job 2 needs resource 1
before(1,0)             // job 1 must be before job 0
before(2,0)             // job 2 must be before job 0
release(0) = 4          // job 0 can be started at time 4 or later
due(0) = 40             // job 0 must be finished by time 40
release(1) = 0          // job 1 can be started at time 0 or later
due(1) = 25             // job 1 must be finished by time 25
release(2) = 0          // job 2 can be started at time 0 or later
due(2) = 30             // job 2 must be finished by time 30
```

Each job needs a single resource exclusively for the full length of the job.

It can be seen that this representation is a generalisation of any job-shop problems. Job-shop problems are sometimes represented as jobs having (sequential or non-sequential) tasks which may require different resources. These can always be decomposed into the above representation.

The Sadeh benchmark suite contains 60 problems, ranging over all levels of difficulty. An easy problem is typically loosely constrained, which means there are many valid schedules, or very tightly constrained, which means there are a few (perhaps just 1) valid schedules while the others are easily discarded as invalid. Difficult problems lie in between, where there may be many schedules which are 'almost' valid, and some that are valid, but it is not easy to distinguish between them heuristically during a search.

2.2 SLACK-based scheduling

The algorithm for SLACK-based scheduling is as follows:

1. establish earliest and latest starting times (est & lst) for each job taking into account the release and due times of the job, as well as explicit ordering constraints specified in the problem (before(1,0)).
2. Create binary variables consisting of job pairs that share a resource and so must be ordered.
3. Search through the space of assignments to these binary variables for an assignment that satisfies all the constraints.

The search then consists of picking a good variable, making an assignment, and then adjusting the est/lst of all jobs affected by the assignment. An invalid assignment is detected if the est/lst of some affected job become invalid (i.e. lst < est).

During the search, we would like pick the maximally constrained variable and then assign to it the minimally constraining value. The 'constraint' for each variable is measured by its slack. Intuitively, this measures the room the two jobs that make up the variable have left to adjust to other constraints (i.e how large is the time gap between est and lst).

3. Representing the job-shop scheduling problem in a Genetic Algorithm

A good representation for a job-shop problem presents many difficulties. The basic problem is that any change in some part of the schedule can totally change the rest of the schedule beyond that point. Also, the partial ordering information in the problem specification (before(1,0)) has to be respected and randomly generating schedules in most representations produce nonsensical schedules.

Any representation that directly encodes start times for jobs runs into immediate difficulties. How do we ensure non-overlapping for jobs sharing the same resource? This may be incorporated in the scoring function by penalizing for overlap and any deviation from the release and due times. But the search space simply has too few valid chromosomes to be effective. Also, there is no good way of penalizing for violation of partial ordering constraints.

A representation similar to the one for SLACK was also considered and tried. In this representation, the est/lst of all jobs is initially calculated as in SLACK. All job pairs sharing a resource are then represented by the chromosome, with each gene representing one particular job pair. Each gene is then binary valued and the chromosome is a binary string.

The string is transformed into a schedule by sequentially assigning to each job pair variable represented by each gene from left to right. After each assignment, the est/lst of all affected jobs are adjusted. Jobs which have an explicit ordering in the problem statement (before(1,0)) are automatically dealt with as they are not part of the chromosome and their est/lst will get adjusted appropriately. However, this approach runs into difficulties too.

Consider the three job pair variables, ab, bc and ac, where a, b and c are jobs. A particular random assignment to these variables may result in a nonsensical assignment such as
a->b (a before b), b->c, c->a. This problem increases exponentially as there are increasing number of jobs that share a resource. Any mutation or crossover can also transform a valid schedule into one containing such 'cycles' and so a nonsensical schedule.

Another representation used in the GA job shop scheduling literature uses a schedule builder to transform the chromosome into an always valid schedule. In this case, the problem is framed as the scheduling of jobs with sequential tasks within each job. Each task requires some resource. The representation then consists of genes which take on values in the set {0,..,N-1} where there are N jobs. There are a total of M genes where M is the total number of tasks in all jobs.

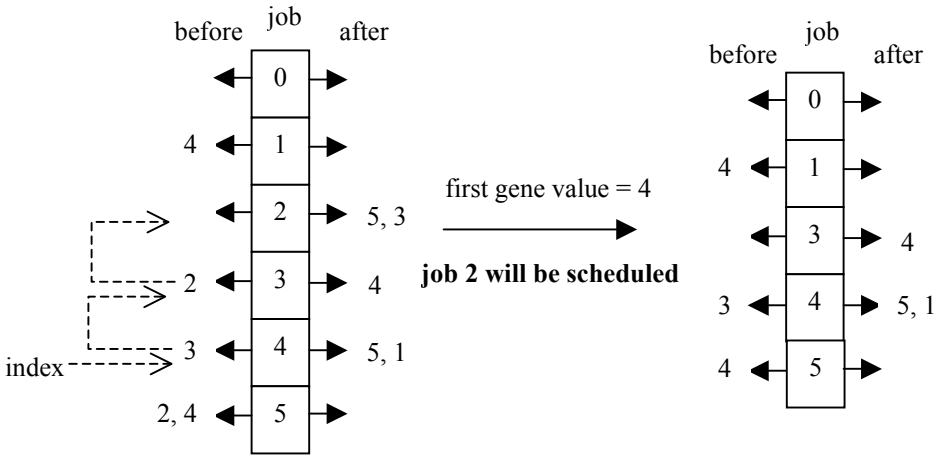
A chromosome *abc…*means: put the first untackled task of the a'th unfinished job into the earliest place it will fit into the schedule. Then put the first untackled task of the b'th unfinished job into the earliest place it will fit into the schedule, and so on. The jobs are numbered in a circular way, so if there are 3 unfinished jobs and the gene's value is 4, job 1 (assuming the jobs are numbered 0,1,2) will be selected. This method ignores any due times for the tasks. These can be incorporated in the score given for each schedule (i.e. a penalty for exceeding the due time for each task or job).

Clearly this approach represents all possible schedules. In addition, it also has the critical property that all schedules are valid. So it is stable under mutation and crossover.

However, the problem it addresses is a simpler one then the one addressed in this paper. All the jobs in the above problem are made up of a total ordering on the tasks for each job. So converting each gene into a job to schedule is easy. The problem addressed in this paper specify a partial ordering on jobs which means that certain jobs have to be done before others.

This problem was addressed by modifying the scheduling algorithm. A list of jobs is built, but with additional information per job. Each job has a before and after list of jobs that are explicitly specified in the problem to be before/after that job. When a gene is translated into a job number, if there is no job on the before list, that job is selected to be scheduled. If there are jobs on the before list, the first job on the list is picked and the same process repeated again. The job that is eventually picked to be scheduled removes itself from the before lists of any remaining jobs.

The diagram below illustrates this process.



jobs list data structure before the first job is scheduled

jobs list data structure after the first job is scheduled

In the above case, the chromosome '402431' would result in jobs being scheduled in the following order: 2, 0, 3, 4, 5, 1.

Each schedule has many different possible representations, but all valid schedules can be represented. This was the representation used in the GA-scheduler built for this experiment.

4. Tableau and scoring function

Objective:	Find the best possible schedule in terms of meeting the problem specifications and the shortest time span
Representation scheme:	Structure = each gene is an integer between 0 and N-1, where there are N jobs. A single chromosome consists of N genes Variable population size and generations (described in the results). Scheduler converts chromosome into schedule as described in section 3.
Fitness cases:	
Fitness:	The timespan of the schedule plus penalties for exceeding due times for any job
Parameters:	population size, generations and penalties varied during the experiment
Termination criteria:	run till number of generations specified, run till timespan = best timespan from SLACK scheduler (explained in the results)

5. Experiment setup, results and discussion

5.1 Experiment Setup

Genesis, an implementation of a GA engine by John Grefenstette, was modified and used as the GA engine. GENESIS has the ability to represent chromosomes as binary strings or floating point number arrays. The number of generations and population size can also be specified, along with the crossover rate, mutation rate and host of other options. The GENESIS engine was modified as required and wrapped with custom code to read the Sadeh problem sets and create the required data structures, as well as set GENESIS up with the required options (range of each gene's values, number of generations, population size, termination criteria if required).

GENESIS requires an evaluation function which will be called for each chromosome that needs to be evaluated. This evaluation function was basically an implementation of the scheduler described above. The score returned was the timespan + any penalties. For example, the total timespan (the completion time of the chronologically latest job) may be 200 time units. For each time unit that any job went over its due

time, a penalty was added to the score. So if the above schedule has a single job that is 5 time units past its due time, 5*penalty_per_unit would be added to the score (of 200 in this case).

The Sadeh problem set was also run with the SLACK algorithm for comparison. The SLACK implementation was modified from an implementation of SLACK by Deshpande, Lim and Sanchez for the CS227 AI course at Stanford. The original implementation simply terminated when it found a satisfactory solution. The modification strived for a better and solution until it ran out of allocated time.

All the code was written in C and the experiments were carried out on the Sun Solaris platform on machines varying from SPARC20's to ULTRA's (in the Sweet Hall computer cluster at Stanford University).

5.2 Results

The results will be described in chronological order in order to motivate each stage of the experiment.

The first task was to get some statistics on the running time and performance of the GA with different parameters on a subset of the problems in order to get optimal parameters with which to run the rest of the problems.

The majority of time in a genetic algorithm run is taken in evaluating each chromosome. So the runtime of a GA would be expected to be linear in the number of generations * population size. This is confirmed by the tables below. (Note: all runtimes are the actual processor time of the job)

population size	number of generations	time (ms)
100	10	330
1000	10	3370
10000	10	34120
100000	10	339040
1000000	10	3397350

Table1: run time information for a single problem with fixed population size and varying population size

population size	number of generations	time (ms)
100	10	310
100	100	3080
100	1000	30670
100	10000	307220
100	100000	3065180

Table2: run time information for a single problem with fixed population size and varying number of generations

It is also important to find a good combination of parameters to test the problem set with. As each schedule must be solved in reasonable time, there is a trade-off between having a large population or a large number of generations. Five problems were then selected and the GA-scheduler run with varying combinations of parameters. The scores for each combination are given in the table below.

		crossover rate / mutation rate								
		0.1 / 0.01	0.2 / 0.01	0.4 / 0.01	0.1 / 0.02	0.2 / 0.02	0.4 / 0.02	0.1 / 0.05	0.2 / 0.05	0.4 / 0.05
p	problem 15									
o	100 / 10000	150	150	146	145	150	143	153	151	149
p	1000 / 1000	150	145	154	148	155	154	154	154	156
u	10000 / 100	152	153	155	150	156	158	151	155	154
l	100000 / 40	147	151	149	153	155	153	154	150	153
a	1000000 / 20	150	151	150	149	151	151	148	148	148
t										
i	problem 28									
o	100 / 10000	150	150	150	150	150	150	150	150	150
n	1000 / 1000	150	150	150	150	150	150	150	150	151
	10000 / 100	150	150	150	150	150	150	150	150	150
s	100000 / 40	150	150	150	150	150	150	150	150	150

i	1000000 / 20	150	150	150	150	150	150	150	150	150
z										
e	problem 35									
	100 / 10000	155	155	155	161	158	159	155	161	161
/	1000 / 1000	159	159	155	156	155	160	162	165	162
	10000 / 100	162	163	161	162	163	162	165	165	165
n	100000 / 40	163	159	159	162	161	158	163	163	158
u	1000000 / 20	161	161	161	161	160	158	161	161	161
m										
	problem 47									
g	100 / 10000	146	144	143	143	144	144	143	144	144
e	1000 / 1000	143	143	145	144	144	145	144	144	144
n	10000 / 100	144	144	146	145	144	144	146	144	145
e	100000 / 40	145	144	145	144	143	144	144	146	146
r	1000000 / 20	143	144	144	143	144	143	143	144	143
a										
t	problem 53									
i	100 / 10000	159	159	166	168	166	168	161	167	172
o	1000 / 1000	166	161	168	165	167	162	171	173	171
n	10000 / 100	168	169	172	176	168	170	168	172	179
s	100000 / 40	171	171	172	168	171	166	171	167	170
	1000000 / 20	169	162	169	167	170	169	170	167	167

Table 3: scores for 5 selected problems for different parameter values

The data above shows that small populations evolved over a large number of generations do better then the other options. Also, the data indicates that having small values for the crossover rate and mutation rate for the small population / large generations is a good combination while having larger values for the crossover rate and mutation rate for large populations / small generations is a good combination. This makes sense theoretically as it is better to evolve slower if there are a lot of generations to evolve, while evolution must take place at a faster rate if there are less generations but a larger population.

Another investigation important for choosing the optimal set of parameters is their effect on runtime. The table below shows the runtime and the number of chromosomes evaluated for three different values of each parameter.

		mutation rate		
		0.01	0.02	0.05
crossover rate	0.1	419112 / 925625	555734 / 993099	995527 / 999001
	0.2	419316 / 933732	552769 / 993828	1034281 / 998999
	0.4	417920 / 950061	557771 / 995049	1009115 / 999000

Table 4: [runtime (ms) / number of chromosomes evaluated in total] over different crossover rate and mutation rate values for a single problem with population size = 1000 and generations = 1000

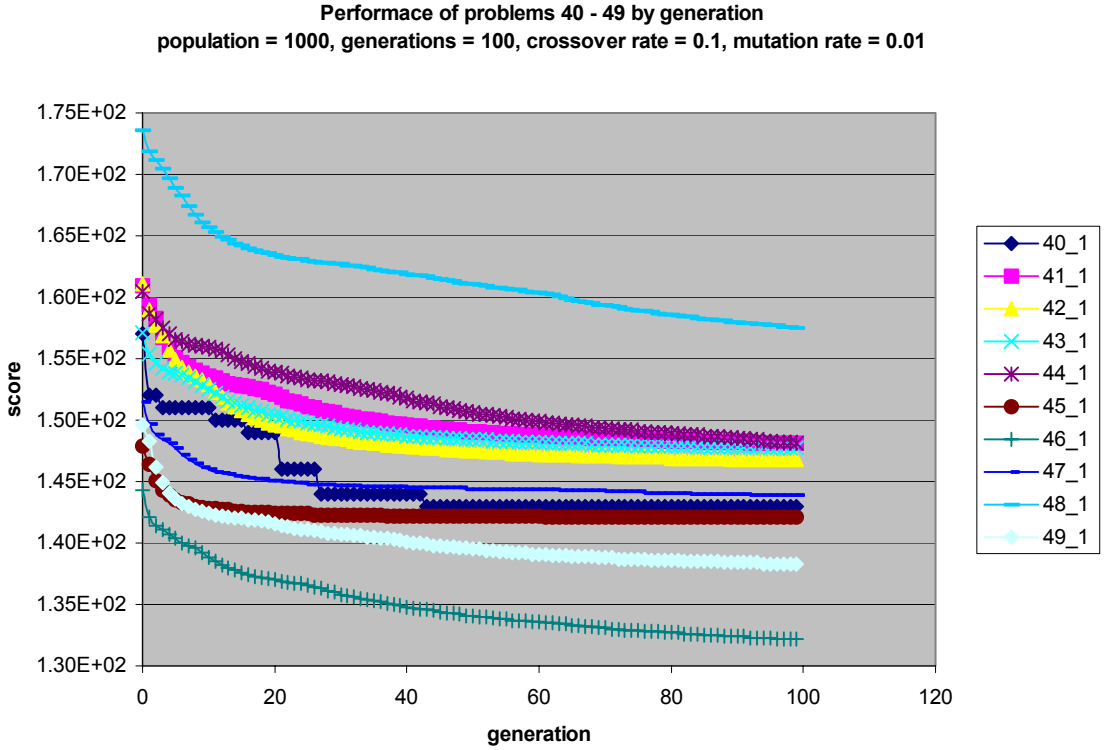
The table shows the runtime for high mutation rates to be considerably higher in comparison to low mutation rates. The runtime for a mutation rate of 0.05 is almost double that of a mutation rate of 0.02 which is itself 20% higher then for 0.01.

Initially, the explanation for this was thought to be a higher number of chromosomes being evaluated. GENESIS only evaluates chromosomes that have changed from generation to generation. So higher crossover and mutation values will increase the number of chromosomes that change and need to be evaluated. But it can be seen that even with the crossover rate = 0.1 and mutation rate = 0.01, 92.6% of all possible chromosomes are already being evaluated. This number goes up to 99.9% when the crossover rate = 0.4 and mutation rate = 0.05. But this increase does not explain the doubling of runtime.

Unfortunately GENESIS could not be profiled (using gprof) to analyse why higher mutations were increasing runtime so much. Most probably it was the actual generation of mutations which was the cause for the performance degrade. At a mutation rate of 0.05 and with 50 jobs per problem, each chromosome has 96% chance of undergoing mutation, while at 0.02 the chance is 64% and with 0.01 there is a 40% chance.

Note that GENESIS also has an 'elitist' mode, in which the best chromosome in each generation is guaranteed to be present in the next generation. This mode was used throughout the experiment.

Another interesting statistic is the convergence rate of the GA-scheduler. Ten problems were chosen and the best schedule at each generation recorded. Graph 1 below shows the results. It can be seen that a couple of problems have converged (indicated by the flat line), while others still have a distinct downward slope at 100 generations when the runs were terminated. Also interesting is the shape of the curves. All but one of the curves show the typical shape. But one of the problems shows the score is minimized in a series of steps.



Since some problems clearly level off quite quickly, it would be efficient to stop the GA once it was detected that the best score had not changed over many generations. However, with a problem that exhibits 'step-like' behaviour as above, this technique may cause the GA to end prematurely.

Another parameter for these problems was the penalty for exceeding the due time on any job, since the GA-scheduler does not intrinsically respect due times in building the schedule. Various values were tried. As expected, having a 0 penalty produced better schedules for some of the problems, since the due times could be ignored. However, once a reasonable penalty was imposed (anything from 10 upwards), all the due times were respected and the value of the penalty did not make a difference (as long as it was above 10).

Having obtained the best set of parameters for the GA-scheduler, the comparison between the GA-scheduler and the SLACK based scheduler was carried out. The ideal parameters for the GA-scheduler were: *population size: 1000, generations: >1000, crossover rate: 0.2, mutation rate: 0.0, penalty = 50.* The number of generations should ideally be as high as possible.

All 60 problems were run with the above parameters, setting the generations to 10000. The problems were split into batches of 10 and run in parallel on different machines. Unfortunately results for 40 problems were lost. These 40 problems were re-run with a slightly different setting (detailed later).

All 60 of the problems were run with the SLACK scheduler, each for 35 minutes. The scheduler was originally designed to simply find a satisfactory solution (i.e. a solution that satisfied all the constraints). This was modified so that after a solution was found, the scheduler internally modified the problem so a better solution would need to be found (by changing any due times of any job that were after the solutions end time to end time – 1). This was done repeatedly until the 35 minute execution time was reached. Each new (and necessarily better) schedule found was marked with the execution time till that point. Once execution finished, the last schedule found was taken as the final result along with the time at which it was found.

The results for the 20 problems run with GA-scheduler (generations = 10000) and the SLACK scheduler are given below.

	GA				SLACK	
problem number	score (timespan of best schedule)	first generation score was reached	time when score was first reached (ms)		score	time when best solution was found
40	143	263	81869		143	420
41	147	333	103659		147	7530
42	145	151	47005		149	70
43	147	76	23658		147	330
44	144	2932	912699		143	71180
45	142	42	13074		142	80
46	130	347	108017		130	250
47	143	363	112998		143	1700
48	153	977	304129		153	160
49	137	417	129808		137	70
50	167	4705	1464615		161	8820
51	153	9347	2909618		148	18180
52	183	2033	632851		170	43540
53	159	3231	1005775		157	690
54	155	9127	2841135		148	43870
55	161	9561	2976234		161	490
56	168	7222	2248129		169	39070
57	168	2183	679544		157	4180
58	171	95	29572		162	90
59	173	2453	763592		166	1200

Table5: Comparison of 20 problems run with GA-scheduler and SLACK

Some interesting observations can be made from the results above. First, both GA-scheduler and SLACK find the same problems easy or difficult. This is only to be expected but is confirmed by the results. SLACK is clearly a much better scheduler for these problems, finding better schedules in much less time. The runtime difference is stark; GA-scheduler takes anywhere between 10 to 1000 times more time and still does not, in many cases, come up with as good a solution.

The number of generations taken by GA-scheduler to reach its best schedule is also interesting. Remember that all of the 20 problems above were run for 10000 generations. So the problems where GA-scheduler and SLACK come up with the same solution fairly quickly, it may be a reasonable assumption that it is in fact the best solution. For other problems where SLACK does better, GA-scheduler reaches a plateau (in terms of the score) at various different times in the search. For instance, in problem 58, GA-scheduler has the score of 171 in the 95th generation, but fails to improve in the 9905 generations that follow, even though

SLACK has found a much better solution fairly quickly. In problem 54 however, it may be that further search may yield a better solution for the GA, having reached its optimal in the 9127th generation.

Due to time limitations, the other 40 problems could not be run again for 10000 generations, but they were run with a slightly modified algorithm. The number of generations was set to 10000, *but the search was terminated if the GA-scheduler matched the SLACK score*. This would provide enough data for a comparison with the SLACK scheduler.

	GA				SLACK	
problem number	score (timespan of best schedule)	first generation score was reached	time when score was first reached (ms)		score	time when best solution was found
0	140	12	3735		140	640
1	143	1	311		143	19790
2	140	17	5292		140	70
3	149	2	623		could not solve	
4	137	77	23969		137	450
5	138	1	311		138	810
6	127	23	7160		127	249780
7	139	271	84359		139	820
8	144	38	11829		144	570
9	131	10	3113		131	310
10	160	1711	532615		157	760
11	147	1061	330278		141	2420
12	168	346	107706		167	200
13	162	5184	1613722		157	20180
14	143	7760	2415603		141	380
15	150	1717	534483		143	770
16	149	9744	3033200		146	500
17	154	2769	861959		151	180
18	155	2816	876590		145	11200
19	165	2408	749584		163	115590
20	140	89	27705		140	11970
21	145	83	25837		145	390
22	143	206	64126		143	25810
23	146	11	3424		146	98330
24	139	52	16187		139	580
25	140	8	2490		140	240
26	129	252	78445		129	8430
27	141	484	150664		141	310
28	150	66	20545		150	340
29	132	613	190820		132	32770
30	161	8078	2514593		158	730
31	147	2135	664602		147	80
32	176	1417	441097		167	1440
33	166	1628	506778		152	7200
34	155	7365	2292643		143	76770
35	155	889	276736		153	9460
36	155	9251	2879735		155	76270
37	164	6327	1969526		154	230
38	165	4788	1490452		152	45380
39	171	188	58522		169	300010

Table 6: Problems 0 – 39 with population size = 1000, crossover rate = 0.2, mutation rate = 0.01, and variable generations.

Note that where the scores of the GA-scheduler and SLACK are equal, the GA-scheduler was terminated and the so did not proceed beyond the 'first generation score was reached'. So, all the white rows were terminated at the first generation that the score matched SLACK's score. However, all of the dark rows ran for 10000 generations because they could not match the SLACK score. The 'first generation score was reached' column for them reflects the first generation that the best score was encountered. So, problem 39 reached a score of 171 at generation 188. It then ran for another 9812 generations but could not find a better schedule.

Although the results above lead to the same conclusions as those from the previous set of 20 problems, note that GA-scheduler managed to solve a problem that SLACK could not solve.

Many of the problems show GA-scheduler reaches its minimum value quite quickly, but then does not improve over many generations. To try and change this, the GA-scheduler algorithm was altered to increase the crossover rate and mutation rate (upto some pre-specified maximum) if there was no change in the score for many generations. Various experiments were run with this 'reverse simulated annealing' type approach, but they did not perform well. However, this technique was only explored in a very rudimentary way and the results are by no means conclusive.

6. Conclusion

Due to representational difficulties, GA's are not able to efficiently find schedules. Crossover and mutation totally change the whole schedule in any representation, thus not following the ideal of preserving most or all of the information contained in the separate pieces of chromosome crossed. Each evaluation of the chromosome also involves building a schedule, a time consuming process which makes GA-schedulers uncompetitive with solutions customized for the particular type of problem, in this case SLACK.

The flexibility of the GA is illustrated by the use of the penalty. In the GA-scheduler, it is very easy to change the scoring function to relax (or somewhat relax) certain requirements to find approximate solutions, something that is not as easy to do in customized algorithms.

7. Future Work

The most important work in improving GA-schedulers is to find a better representation that will make crossover and mutation more meaningful. Also, it is not clear why the GA-scheduler remained at the same score for many generations. This should be investigated further and perhaps solutions like 'reverse simulated annealing' investigated in greater depth.

References:

David E Goldberg. 1989. Genetic Algorithms in Search, Optimization & Machine Learning. Addison Wesley

Bierwirth, C, Mattfeld D. 1999. Production Scheduling and Rescheduling with Genetic Algorithms. Evolutionary Computation 7(1): 1-17, MIT.

Fang, Ross, Corne. 1993. A Promising genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling and Open-Shop Scheduling Problems. Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, pages 375-382

Yamada T, Nakano R. 1997. Genetic Algorithms for Job-Shop Scheduling Problems Proceedings of Modern Heuristic for Decision Support, pp. 67-81, UNICOM

Jain A, Meeran S. 1998. A State-Of-The-Art Review of Job-Shop Scheduling Techniques Department of Electronic and Mechanical Engineering, University of Dundee, Scotland, UK