Using Evolutionary Algorithms for Job Shop Scheduling

Sam Serrels
40082367@napier.ac.uk
Edinburgh Napier University
Computational Intelligence (SET10107)

Abstract

This report documents the implementation and tuning of an Evolutionary Algorithm in Java to solve a set of Job Shop Scheduling Problems. Computing optimal solutions for JSS problems in an NP-Hard task, therefore a brute-force approach is out of the question for problems with large sets of jobs. Evolutionary approaches have been proven to produce good and even optimal solutions in a reactively amount of time, this report implements an EA method using components describes by the literature, and then analyses the solutions produced and the different methods of reaching them.

1 Introduction

JSSP A Job Shop Scheduling Problem is made up of a set of Machines and Jobs, each Job contains a set of operations that must be processed in order and sequentially. The Operations must be processed by a specific machine, each machine can only process one operation from one job at a time. Each Operation will utilise a machine for a pre-determined amount of time A solution to the problem takes the form of a list stating the order of operations for each machine so that all jobs are completed in the shortest amount of time possible. This report uses a common simplified version of JSSP wherein each Job has the same amount of operations as there are machines, so that every job will use every machine exactly once.

Encoding for Genetic Evolution A solution to a JSSP problem is implemented in code as a 2D array, with each row representing a machine, and each column representing a Job, ordered by the order in which they should be processed. It should be noted that a machine processes an operation, not a job, but as each job only has one operation per machine, the corresponding job can be found trivially and so Job IDs can be used to simplify the implementation. The encoding does not encode the time of execution, only the order. It is the job of the fitness function when decoding the solution to determine *when* each operation is processed, abiding by the operation prerequisites and machine occupancy constrains.

Evolving a Solution The rows of a solution array form the chromosomes to be processed by the evolutionary algorithm. Each row is a set of all the available Job IDs, and therefore any permutation of this set is a valid solution.

Comparing solutions To obtain a fitness score for a solution, it must be decoded into a final schedule. This process involves determining the start and end time for processing operations on each machine. To do this, the order of operations in the solution is followed for each machine until an operation cannot be scheduled due to a non-complete pre-requite operation, in this case the machine will sit idle until the operation can be processed. Once this process is finished, the total runtime can be determined and used as a fitness value. Other metrics could also be used for fitness (e.g maximum Idle time, or concurrent machine occupancy), but this report only uses runtime to compare solutions.

2 Previous Work

Using GAs to solve the JSSP has been documented in multiple papers, each taking a different approach to optimise different parts of the GA process. As this is a permutation based problem with multiple constraints, the solutions which provide algorithms that are more tailored to fit the specifically JSSP rather than a general permutation problem seem to get the best results.

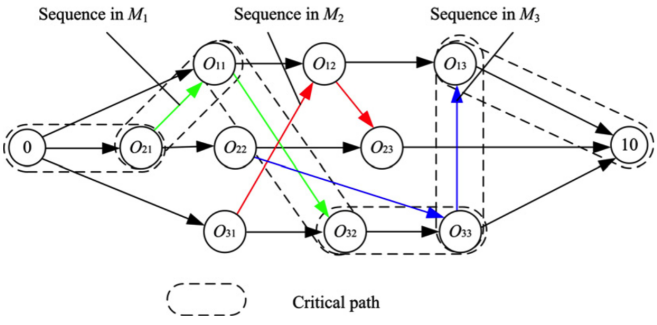

Figure 1: ***Disjunctive graph of a solution*** *- [Gao et al. 2011]*

Visualising Solutions A common technique taken by many articles is to represent JSSP solutions visually using a Disjunctive graph, this can communicate the order of operations within a job, which machine an operation is on, the order of operations for each machine, and also the critical path.

Chosen Reference Paper This report closely follows the work by Ren Qing-dao-er-ji and Yuping Wang in their paper "A new hybrid genetic algorithm for job shop scheduling problem" [dao-er ji and Wang 2012]. This report closely follows the structure of the documented GA with additional algorithms described in other papers.

Concentration based selection The paper defines a complex method of Gauging the similarity between two solutions, this is used in conjunction with a fitness value when selecting form the population for crossover. This is used to reduce early convergence.

Critical Path Mutation A method of mutating a solution based on the critical path was used, this is documented later (section 3.5), this specific algorithm was out of scope for this project but would have been the first to implemented in future work as the results provided in the paper seem very promising.

Local Search The paper describes using a limited local search operator after mutation to increase the quality of the new child solutions, this was implemented in this project and it was found to be highly beneficial to the performance of the GA.

Results The results shown in the paper are some of the best in the field, other attempts have been shown alongside for comparison. A criticism is that (along with many other papers on this subject) only the generation count is recorded, without any time measures. The specifications of the hardware are given, but not how long the proposed total algorithm will take.

3 Methodology

The method for designing an effective GA was to implement and experiment with a section of different Crossover/Mutation/GA structure/Solution Generation techniques, some from the literature and some original for this report.

Library Use The Implementation was built on top of a provided utility library for generating random solutions, it was also used for solution validating and fitness calculation.
The Apache HttpComponents Library, and Google's Gson Library were used in the network distribution code

Implementation of Literature A mixture of various functions and techniques described by the literature were implemented and the effects of combining them into a working genetic algorithm was recorded.

3.1 Solution Quality Metrics

As the Genetic algorithm is running, a method was needed to easily gauge progress and to spot problems. This was achieved by calculating the average fitness of various percentile sections of the population, and printing this to the output every generation along with other statistics like the current best fitness and best all time fitness. Divergence is calculated by comparing the average fitness of the entire population compared to same average of the last generation. See Figure 7 for a sample output.

3.2 Initial population generation

A theory that was tested was that if the initial seed population was of a higher quality than a pure random population, then this would increase the efficiency of the overall evolution. Three new algorithms were created and another which generated random solutions and then optimised them with the Feasibility Algorithm were tested.

Job Runtime Priority This generation technique ordered jobs by their total runtime, and gave scheduling priority to the longest running jobs. The theory behind this was that re-arranging smaller operations would always cause less disruption than moving large operations, so it may be beneficial to have the longest running operations in their most optimal place as soon as possible.

Operation Order Priority This generation technique attempted to place the first operation of all job first, followed by the second operation of all jobs until all operations were placed. The theory behind this was to aim to reduce machine idle time by reducing instances where operations were scheduled before there prerequisite operations were finished. This follows a similar concept to the feasibility optimisation algorithm, but without attempting to fix dependencies.

Best Random / Best Hybrid This solution generator would generate a pool of either random solutions, or solutions created by other generators and return the solution form the pool with the best fitness. The pool size could be adjusted, but it was found that a size of 128 worked best.

Feasibility Optimised In the paper by [dao-er ji and Wang 2012], a solution is considered feasible if every operation is scheduled either at the same time or after any prerequisite operations on other machines. In the paper, only solutions that are feasible are considered for fitness calculations. This report implements a method that attempts to turn any solution into a feasible solution, however not all problems can have a feasible solution, and not all solutions can be made 100% feasible. If only

feasible solutions are considered, then the total set of possible solutions is vastly reduced, however the algorithm in this report maintains enough variance that population diversity is still high enough for genetic evolution.

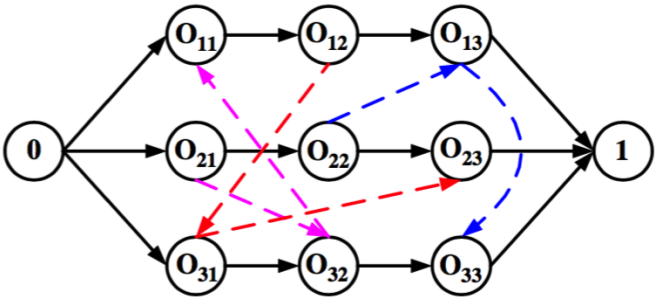

Figure 2: ***Disjunctive graph of an Infeasible solution*** *- [dao-er ji and Wang 2012]*

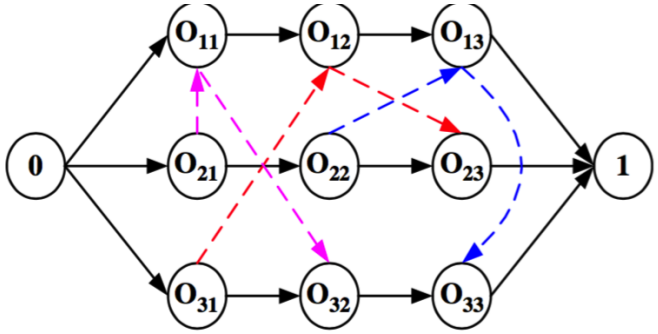

Figure 3: ***Disjunctive graph of the fixed feasible solution*** *- Notice how the operation order of the machines never heads to the left[dao-er ji and Wang 2012]*

This solution uses a modified version of the Best Random generator, where each random solution is optimised before comparison.

3.3 Selection

Three different methods of Selection were attempted.

Roulette Selection The population is sorted by fitness, the solutions with the best fitness have a higher chance of being selected for crossover.

Experimental Selection The population is sorted by fitness, then each solution is paired with all solutions that have a lower fitness than itself.

Tournament A random selection of the population is taken, this is then sorted by fitness, and then the best solutions in this pool are paired together for crossover.

3.4 Crossover

Five different crossover functions were implemented.

Single and Multi-Point A random machine ID is selected from the two solutions and then a crossover point is selected randomly from the list of operations, everything after this point is swapped over from the two solutions. For Multi-point, two random crossover points are selected and everything in-between them is swapped. This can create invalid solutions as there will be duplicate jobs within the job list, there are many different approaches to fixing this, the method used by this report was to remove the duplicates and fill in the blanks with the missing job operations, ordered by the lowest operation ID.

Machine Swap This method selects an entire set of operations from a machine and swaps it completely with the list from the corresponding machine in another solution. This has the benefit that a solution will always be valid, but the total amount of unique outcomes for two input solutions is limited by the number of machines. This is used in the paper by [dao-er ji and Wang 2012]

Partially Matched Two crossover points are selected and the containing operations are swapped, similar to Multi-Point Crossover. The positions of the operations in each matching swapped sections are used to form a lookup table, this is then used to replace duplicate operations. [Sivanandam and Deepa 2007]

Liang Gao This crossover follows a method described by [Gao et al. 2011]. The set of jobs is split randomly into two sets. Jobs in set 1 are copied into the same position in child A as parent 1, and jobs in set 2 are copied into the same position in child B as Parent 2. Then the empty spaces are filled in according to the order of the opposing parent. Child A fills in the empty spaces according to the order of operations from Parent B, and Child B from Parent A. This is demonstrated in Figure 4. It is possible in certain situations that a child will be identical to the a parent, so this must be checked for if it is an undesirable outcome.

3.5 Mutation

Critical Path Mutation / Local search If the critical path of a solution can be determined, the paper by [Gao et al. 2011] describes a method wherein if two or more operations on the critical paths are placed consecutively on the same machine, then all permutations of the order of these operations is searched for fitness improvements. This was not implemented in this report as the complexity of determining the critical path was out of the scope for this investigation.

Operation Mutation This is theoretically the simplest form of Mutation, a random machine is picked, and then two of the operations within it are swapped. This was the mutation operator used in this investigation.

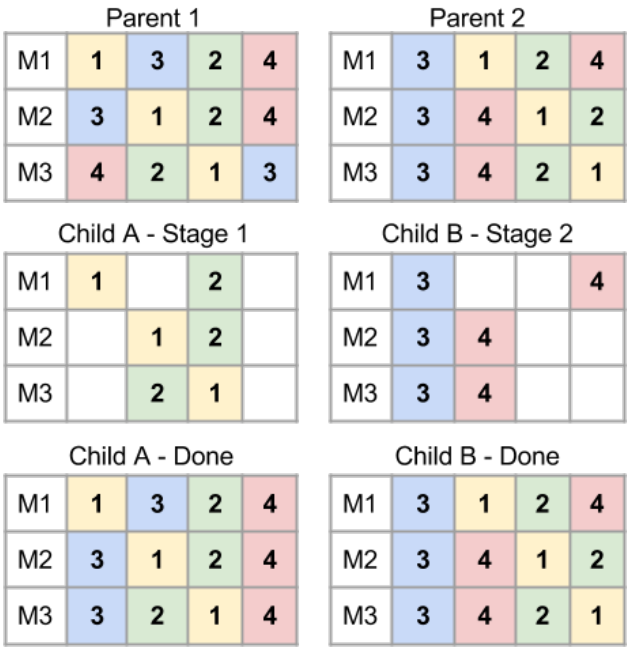

Figure 4: ***Liang Gao Crossover*** *- Notice how Child B is identical to Parent 2*

3.6 GA Structure

There are many large decisions and small nuances to consider when designing a GA, mainly to do with handling the population. The implementation of this report tried out several different GA structures, the difference are detailed below.

Child Placement When new children are generated, do they replace their parents or are they inserted into the population along with their parents? Replacing parents can overwrite good solutions, and keeping parents decreases diversity.

Population Size The size of the population is also an area which can have great effect on the performance fo the GA, too small and there will not be enough diversity, too large and the rate of improvement will drop and the processing time for each generation may become infeasible. If the population size is to be kept constant which members are removed when the size increases?

Mutation The mutation operations could apply only to new children as they are generated, or the entire/subset of the population could be mutated every generation.

Duplicate Solutions This is simply to decide whether duplicate identical solutions are allowed in the population. Ensuring no duplicates is a substantial and time consuming task, on the other hand, clones can quickly reduce diversity and cause early convergence.

Reset operation If the population is seen to converge on a non-optimal solution, then the entire population is mutated and the top solutions are removed. This could be used as a technique to escape from a local maximum without fully restarting the algorithm for scratch. The criterion for initiating a reset could be adjusted beforehand and during the operation of the GE.

3.7 Live Parameter adjustment

Almost all of the parameters within the GA can be altered during runtime, different techniques were tried to couple the live population metrics to the GA parameters, such as increasing the amount of crossovers as overall fitness decreased, or increasing the total population size depending on population diversity.

3.8 Termination Procedure

While running an GA indefinitely until a lower bound is reached is a viable approach, this investigation required additional termination events to enable the analysis of the effectiveness of different GA parameters, in order to find the optimal settings for each different JSS problem. The additional termination events were based on: Total Time taken, Generation Count, and Population Diversity.

3.9 Distribution

To achieve data for the performance of GA parameters for each supplied JSS problem, the program was distributed over a network of computers. Each computer would ask for a set of parameters from a dispatch server, run the GA with these parameters and then return the results.

Network system design For the sake of simplicity the communication protocol used was HTTP, and the data encoding used was JSON. As JSON can easily be converted to and from Java objects. The dispatch server runs a simple HTTP web-server, listening for requests and sending GA parameters (known as WorkOrders in the implementation) to the worker. The worker then returns the result of the GA (WorkResponce) as HTTP POST data. This is stored in a results file by the dispatch server.

Work Management The Dispatch server keeps track of which jobs have been dispatched and not returned, and will remove them from the list if they are not returned within a specific time limit. This means that if a worker crashes or goes offline and does not respond with a result, the same job will be resent to another worker at a later time. Each time a job is dispatched or returned the entire state is saved to disk, this means the dispatch server can be taken down and modified at any time.

Job creation The method for determining the parameters to send to a client could include an entirely separate genetic algorithm within itself. This was out of the scope for this project, so a very basic hill-climber approach was implemented. When an request for a job is received, the list of JSS problems is searched to see which has the fewest jobs completed, this is then passed ot a function which generates GA parameters for this problem. This function initially increases the population size and maximum generation limit for each run completed until completing a run would take too long. It then takes the best run so far and randomly mutates it until a better set of parameters is found.

4 Experiment Plan

The plan for the implementation was to first get the basic structure of a GA running with a basic single point crossover and simple mutation. In these early stage, creating a method for measuring the performance of the GA was crucial as this needed to be implemented before nay improvements could be made.
Once the performance measures are in place, additional algorithms can be implemented, as described in the Methodology section. The performance should be tuned based on a small set of problems initially for quick iteration times, problems 86, 101, and 95 were chosen for use during the implementation stage.
After the specified functions and algorithms were implemented, the code should be analysed and experimented upon to increase performance and solution quality.
After the algorithm has reached an adequate level of performance, different problems can be tested and the optimal parameters for each problem will be searched for. This task will be distributed over a network of computers to gather results quickly. The parameters for each problem will start with the smallest and quickest settings and then build up over time to allow the GA to run longer with larger populations and search spaces. Once the best set of parameters is found, this will mutated pseudo randomly in a local search fashion to try and gain small improvements.

5 Results

5.1 Population Generation

Using both Job runtime and operation order priority population generation did give a significant boost to the fitness of the first generation. However the amount of possible solutions they can generate is limited and thereafter they must fall back to randomness and therefore the benefit of their use is limited and does not scale.
Using Feasibility optimised solutions for the initial population gave an extreme boost to solution quality, which carried over into many generation in the GA. For some problems the lower bound would be found in the initial population without running the GA at all. The algorithm is expensive to run, but the benefits vastly outlay this cost. A visual representation of the results of the algorithm is shown in Figure: 5 & Figure: 6 .

5.2 Feasibility Optimisations

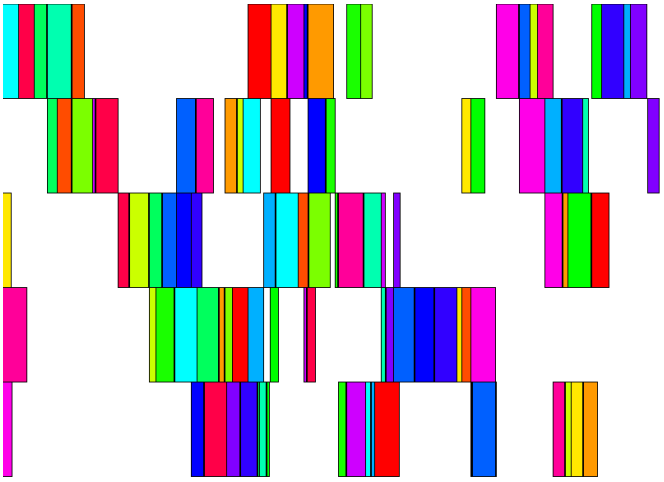

Figure 5: ***Problem 100, Random Solution*** *- Fitness 2471*

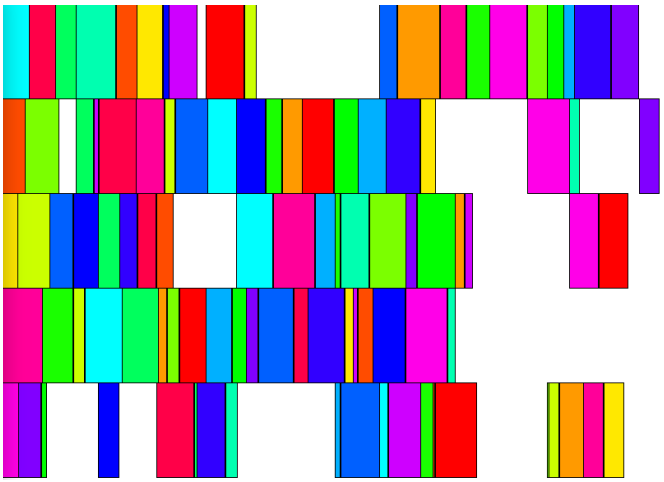

Figure 6: ***Problem 100, Random Solution, Optimised*** *- Fitness 1496*

5.3 Crossover

It was observed that the crossover had the least effect on improving solution when compared to the mutation stage, generally crossovers that significantly changed the solution were overall damaging to fitness improvements. The Partially Matched crossover performed well but had a high processing time, while the Liang Gao algorithm worked quickly and tended to leave the fitness roughly the same or slightly better.

5.4 Mutation

Operation mutation worked well in testing and as this is the smallest possible change that could be made to a solution it was theorised that altering the mutation operator would not be a worthwhile use of time.

5.5 GA structure

Changing the structure of the GA provided the most significant changes to the rate of improvement and convergence. The main issue was the convergence on local maximums, this is the reason for the use of the reset operation which worked well but did not fix the source of the problem.
Changing the GA parameters on the fly to increase diversity as a combative measure when the convergence rate rose did not prove effective as at this point the population was already on track to converge.
Using a small tournament to pick parents, and then a small local search after the mutation operator, coupled with a function that guarantees a mutated child is not a clone of any other solution in the population was by far the most effective version of the GA found in this investigation.

5.6 Parameter Searching

The distributed approach had many benefits, primarily an increase in productivity as it allowed for quickly testing out new theories easily while at the same time gathering thorough results for any previous ideas. This highlighted bugs in the GA code much quicker which saved a large amount of time as every time a bug was found, the entire results set may have to be wiped and restarted.
Large problems had a tendency to time-out as a single generation could take as long as 40 minutes at times, this would mean that the dispatch server would think that the worker had gone off-line and re-distribute the job, and then run into the same issue on another machine. The distribution logic was constantly monitored, tweaked, and fixed throughout the experiment period. Using a network of 100 High-Spec desktop computers, each running 8 instances of the algorithm over the period of roughly a week, this project accumulated around 100'000 compute hours finding optimal GA parameters (Including several total restarts).

6 Conclusions

This report concludes that when it comes to a specialised permutation based problem like JSSP, generic GA algorithms tend to be more destructive than constructive, the more logic than an algorithm has that is tailored to the specific problem the more effective it will be. Furthermore, in permutation problems in general, small mutation functions and local searches are much more powerful than the crossover operations.
Spending time to optimise the initial population can be extremely beneficial for certain problems, and is therefore worth the implementation and processing time.
Maintaining diversity is key to avoiding convergence, and new fresh random solutions should be introduced into the population where possible to maintain diversity. In the case of this investigation, any duplicate was replaced with a completely random solution.
The results achieved by this investigation are reasonably close to the known lower bound for the given problems, although there are examples in the literature that perform better. Overall the key takeaway from this investigation is that there are many similar yet different approaches to solving JSSPs with Genetic Algorithms, the effectiveness of each vary wildly and only certain fine tuned combinations of algorithms can do well across a wide range of problems. Even then, the perfect parameters to input into the tuned GA is an entirely separate and equally large task in itself.

References

dao-er ji, R. Q., and Wang, Y. 2012. A new hybrid genetic algorithm for job shop scheduling problem. *Computers & Operations Research 39*, 10, 2291–2299.

DRISS, I., MOUSS, K. N., and LAGGOUN, A. 2015. An effective genetic algorithm for the flexible job shop scheduling problems.

Gao, L., Zhang, G., Zhang, L., and Li, X. 2011. An efficient memetic algorithm for solving the job shop scheduling problem. *Computers & Industrial Engineering 60*, 4, 699–705.

González Fernández, M. Á., Rodríguez Vela, M. d. C., and Varela Arias, J. R. 2013. An efficient memetic algorithm for the flexible job shop with setup times. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, Association for the Advancement of Artificial Intelligence (AAAI).

Sivanandam, S., and Deepa, S. 2007. *Introduction to genetic algorithms*. Springer Science & Business Media.

Zhang, G., Gao, L., and Shi, Y. 2011. An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications 38*, 4, 3563–3573.

7 Appendix

```
Run:1 BestEver: 2248 Top:2248 avg10:2345 avg25:2390 avg50:2423 avg:2464 improvement: -53 divergence:-74 divavg:246
Run:2 BestEver: 2224 Top:2224 avg10:2317 avg25:2358 avg50:2391 avg:2430 improvement: -28 divergence:-34 divavg:243
Run:3 BestEver: 2208 Top:2208 avg10:2306 avg25:2345 avg50:2376 avg:2416 improvement: -11 divergence:-14 divavg:241
Run:4 BestEver: 2182 Top:2182 avg10:2280 avg25:2322 avg50:2353 avg:2394 improvement: -26 divergence:-22 divavg:239
Run:5 BestEver: 2150 Top:2150 avg10:2261 avg25:2302 avg50:2333 avg:2374 improvement: -19 divergence:-20 divavg:237
Run:6 BestEver: 2145 Top:2145 avg10:2205 avg25:2243 avg50:2270 avg:2305 improvement: -56 divergence:-69 divavg:230
Run:7 BestEver: 2128 Top:2128 avg10:2200 avg25:2239 avg50:2266 avg:2301 improvement: -5  divergence:-4  divavg:230
Run:8 BestEver: 2123 Top:2123 avg10:2158 avg25:2187 avg50:2208 avg:2236 improvement: -42 divergence:-65 divavg:223
Run:9 BestEver: 2095 Top:2095 avg10:2156 avg25:2185 avg50:2206 avg:2234 improvement: -2  divergence:-2  divavg:223
Run:10 BestEver: 2094 Top:2094 avg10:2150 avg25:2178 avg50:2197 avg:2223 improvement: -6 divergence:-11 divavg:-31
Run:11 BestEver: 2078 Top:2078 avg10:2142 avg25:2171 avg50:2189 avg:2215 improvement: -8 divergence:-8  divavg:-24
```

Figure 7: ***Output from EA*** *- Problem 119*

```
Job 6640271793819351040Took too long to return, resettting
Job 3969059755470964736Took too long to return, resettting
Job 3589097351969904640Took too long to return, resettting
Job 7288509481955014656Took too long to return, resettting
Jobs in-flight: 497, completed jobs: 384
data Received from: /146.176.165.140:6188
WR 8970837455698326528 returned, stall score: 1836 (1213) gen:28 (12) Time:1100262 pid:20 128
Dispatching Job: 8817262558380635136 _ 106 _ /146.176.165.140:6189
data Received from: /146.176.165.91:1172
WR 8069680808697445376 returned, stall score: 1737 (1211) gen:48 (42) Time:2094005 pid:16 178
Dispatching Job: 6531524674258251776 _ 110 _ /146.176.165.91:1173
data Received from: /146.176.165.94:1172
WR 8868915356402914304 returned, stall score: 1837 (1213) gen:21 (12) Time:2111326 pid:20 128
Dispatching Job: 8341506546131312640 _ 120 _ /146.176.165.94:1173
data Received from: /146.176.165.137:56219
WR 7588465146849695744 returned, stall score: 1841 (1213) gen:36 (12) Time:1779302 pid:20 128
Dispatching Job: 610655564806776832 _ 135 _ /146.176.165.137:56220
data Received from: /146.176.165.84:60181
WR 5699269066937261056 returned, stall score: 3620 (2781) gen:18 (18) Time:521660 pid:56 228
data Received from: /146.176.165.20:7608
WR 1445383608537304064 returned, stall score: 3952 (2679) gen:8 (8) Time:599865 pid:55 228
Dispatching Job: 1709350564232275968 _ 14 _ /146.176.165.84:60182
Dispatching Job: 8476883259144888320 _ 15 _ /146.176.165.20:7609
data Received from: /146.176.165.234:3538
WR 2239092689186534400 returned, stall score: 3731 (2813) gen:3 (3) Time:682486 pid:54 328
Dispatching Job: 6242727016341753856 _ 17 _ /146.176.165.234:3539
data Received from: /146.176.165.28:7386
WR 1181199714267614208 returned, stall score: 1816 (1213) gen:30 (12) Time:2555143 pid:20 128
Dispatching Job: 7785820994601106432 _ 16 _ /146.176.165.28:7387
```

Figure 8: ***Console output snippet from Dispatch Server*** *- Currently talking to 150 Clients*

	Goal	Size	Best Score	Gen	Pop Size	Total Time		Goal	Size	Best Score	Gen	Pop Size	Total Time
1	1005	15X15	1595	17	1228	21.26	72	5181	100X20	6404	154	227	27.89
2	953	15X15	1452	24	128	3.47	73	5552	100X20	6765	192	206	22.47
3	1036	15X15	1487	21	418	20.41	74	5339	100X20	6537	146	153	26.69
4	973	15X15	1687	14	428	20.44	75	5392	100X20	6656	159	187	29.55
5	940	15X15	1615	30	336	20.41	76	5342	100X20	6598	174	278	50.97
6	1134	15X15	1693	32	297	20.29	77	5436	100X20	6681	161	206	21.02
7	1103	15X15	1437	30	739	20.51	78	5394	100X20	6591	108	212	36.78
8	980	15X15	1448	34	1034	20.72	79	5358	100X20	6606	146	196	26.21
9	1020	15X15	1744	22	1223	21.18	80	5183	100X20	6402	111	136	24.81
10	940	15X15	1613	45	939	20.89	81	872	10X10	1411	9	528	20.16
11	1254	20X15	1805	45	323	20.37	82	742	10X10	1044	10	1220	20.36
12	1267	20X15	1703	44	328	20.40	83	578	20X15	791	65	562	20.86
13	1243	20X15	1557	61	489	20.63	84	566	20X15	835	66	237	20.41
14	1329	20X15	1615	66	343	20.62	85	563	20X15	820	54	537	20.57
15	1163	20X15	1608	84	245	20.37	**86**	55	6X6	55	3	541	20.03
16	1211	20X15	1664	58	419	20.52	87	718	10X10	1086	18	278	20.08
17	1306	20X15	1810	61	677	21.38	88	1164	20X5	1283	27	678	20.18
18	1315	20X15	1697	57	533	20.71	89	666	10X5	678	4	178	20.02
19	1202	20X15	1645	66	480	20.50	90	655	10X5	725	5	1064	20.14
20	1213	20X15	1808	26	314	20.39	91	588	10X5	650	6	626	20.07
21	1217	20X20	2130	61	239	20.75	92	567	10X5	698	2	528	20.06
22	1314	20X20	2111	55	409	20.75	**93**	593	10X5	593	0	128	0.01
23	1248	20X20	1879	54	135	20.22	**94**	926	15X5	926	13	131	0.03
24	1284	20X20	2222	35	278	20.82	95	869	15X5	922	10	1170	20.23
25	1256	20X20	2049	83	214	20.64	**96**	863	15X5	863	9	119	0.14
26	1245	20X20	2009	99	337	21.01	97	951	15X5	955	6	678	20.11
27	1403	20X20	2216	97	451	20.89	**98**	958	15X5	958	0	128	0.03
28	1387	20X20	2188	59	440	22.04	99	1222	20X5	1253	8	521	20.13
29	1352	20X20	1962	79	294	20.59	**100**	1039	20X5	1039	0	128	0.04
30	1277	20X20	2082	68	285	20.52	101	1150	20X5	1150	10	78	0.03
31	1764	30X15	2214	108	247	20.39	**102**	1292	20X5	1292	0	128	0.03
32	1774	30X15	2216	129	348	20.69	103	1207	20X5	1225	24	1178	20.33
33	1733	30X15	2234	118	167	20.38	104	717	10X10	1014	18	379	20.13
34	1828	30X15	2225	85	170	20.39	105	683	10X10	939	8	365	20.10
35	1754	30X15	2417	69	207	20.76	106	663	10X10	915	12	1169	20.32
36	1777	30X15	2290	77	178	20.44	107	685	10X10	1065	9	628	20.21
37	1771	30X15	2333	60	314	45.72	108	756	10X10	1056	4	121	3.37
38	1673	30X15	2067	95	237	20.56	109	954	15X10	1243	16	567	20.35
39	1764	30X15	2286	78	357	20.88	110	907	15X10	1133	29	709	20.27
40	1608	30X15	2136	79	278	20.89	111	1032	15X10	1118	35	928	20.50
41	1850	30X20	2547	89	272	20.03	112	857	15X10	1067	27	777	20.39
42	1761	30X20	2579	139	171	21.34	113	864	15X10	1291	11	228	20.12
43	1710	30X20	2439	90	180	20.91	114	1218	20X10	1425	41	560	20.40
44	1820	30X20	2611	132	168	20.62	115	1188	20X10	1510	32	928	20.81
45	1785	30X20	2630	90	157	20.48	116	1216	20X10	1371	29	178	20.12
46	1940	30X20	2778	133	168	20.96	117	1105	20X10	1357	47	328	20.26
47	1751	30X20	2470	66	180	20.74	118	1355	20X10	1606	27	178	20.14
48	1770	30X20	2546	76	159	20.58	119	1784	30X10	1923	47	494	20.69
49	1758	30X20	2497	116	217	21.16	**120**	1850	30X10	1850	110	583	20.14
50	1678	30X20	2415	131	149	20.65	121	1719	30X10	1755	60	187	20.53
51	2760	50X15	3416	140	113	20.54	122	1721	30X10	1828	57	425	20.44
52	2756	50X15	3411	171	244	21.32	123	1888	30X10	2202	30	228	20.31
53	2717	50X15	3167	145	187	21.58	124	1028	15X15	1456	41	457	20.42
54	2813	50X15	3209	125	188	19.27	125	1132	15X15	1715	30	758	20.42
55	2679	50X15	3227	133	167	21.01	126	943	15X15	1532	23	712	20.51
56	2781	50X15	3243	139	303	22.11	127	1128	15X15	1558	26	458	20.57
57	2943	50X15	3553	48	178	21.56	128	1055	15X15	1549	41	128	3.47
58	2885	50X15	3343	123	158	20.82	129	928	10X10	1270	10	478	20.13
59	2655	50X15	3209	143	199	21.65	130	733	10X10	1139	13	1178	20.36
60	2723	50X15	3065	140	157	21.01	131	830	10X10	1286	9	1229	20.30
61	2868	50X20	3717	111	154	21.85	132	793	10X10	1102	26	970	20.22
62	2848	50X20	3802	182	258	23.14	133	733	10X10	1323	8	986	20.26
63	2755	50X20	3376	220	214	21.28	134	930	10X10	1385	11	1268	20.36
64	2697	50X20	3457	144	300	22.45	135	326	10X10	492	11	128	3.37
65	2725	50X20	3512	166	168	21.12	136	779	10X10	1260	9	569	20.15
66	2845	50X20	3571	187	223	22.72	137	705	10X10	1135	16	1118	20.27
67	2812	50X20	3788	158	181	20.32	138	783	10X10	1477	12	1233	20.42
68	2764	50X20	3400	132	107	20.82	139	694	20X20	1084	68	378	21.04
69	3071	50X20	3722	131	130	21.27	140	713	20X20	1118	62	292	49.64
70	2995	50X20	3766	166	213	18.17	141	680	20X20	1101	84	391	46.58
71	5464	100X20	6922	136	148	29.15	142	734	20X20	1185	70	226	20.64

Table 1: *Best results for all problems*