MATEMATIKA, 2006, Volume 22, Number 2, pp. 91–107
©Department of Mathematics, UTM.

A Modified Giffler and Thompson Genetic Algorithm on the Job Shop Scheduling Problem

Lee Hui Peng & Sutinah Salim
Department of Mathematics, Faculty of Science, Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia.

Abstract Job Shop Scheduling Problem (JSSP) is one of the well-known hardest combinatorial optimization problems. The goal of this research is to study an efficient scheduling method based on Genetic Algorithm (GA) to address JSSP. A GA based on Giffler and Thompson (GT) algorithm known as GT-GA that utilizes the GT crossover is investigated. This algorithm is modified to produce better results than the existing algorithm by using Visual Prolog programming language.

Keywords Job Shop Scheduling Problem, Genetic Algorithms, GT-GA.

1 Introduction

Genetic algorithm (GA) is an optimization technique for a function defined over finite (discrete) domain. This search algorithm is heavily inspired by the mechanics of natural selection and natural genetics in plants and animals. It is created to mimic some of the processes observed in natural and biological evolution [1]. It combines survival of the fittest among string structure with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search [2]. In every generation, aof artificial creatures (strings) is created using bits and pieces of the fittest of the old: an occasional new part is tried for good measure. We should note that while randomized search was imposed, it does not necessarily imply directionless search. The search efficiently exploits historical information to speculate on new search points with expected improved performance. Therefore it differs significantly from random search, which can be considered as a blind search. The idea of using a population of solutions to solve practical engineering optimization problems was considered several times during the 1950's and 1960's [3].

GA was developed by John Holland, his colleagues, and his students at the University of Michigan in the 1970s [1]. However, GA is finally popularized by one of John Holland's students, David Goldberg, who was able to solve a difficult problem involving the control of gas-pipeline transmission for his dissertation in 1980s [4].

In this paper, we will discuss the Giffler and Thompson Genetic Algorithm (GT-GA) [5] and the modification of this algorithm. On the average we observe that the modified GT-GA can solve the JSSP in shorter time. In addition, the effect of different setting of GA parameter will also be investigated.

The paper is organized as follows: Section 2 describes the existing GT-GA; Section 3 describes a modified GT-GA; Section 4 describes the conducted analysis such as the

comparison using benchmark problems and effect of the GA parameters; Section 5 concludes the paper.

2 Giffler and Thompson Genetic Algorithm

The Giffler and Thompson approach to GA for crossover as described by Yamada [5] is outlined in the following algorithm. The crossover provides a method whereby information for differing solutions can be melded to allow the exploration of new parts of the search space, and the mutation serves to maintain diversity in the population by making random changes to single element of the string. The terminology such as chromosome is a string representing a solution to the problem and population is the set of chromosomes used in a given generation (iteration).

Algorithm 1 A GA using the GT crossover

Note:	The technological sequence matrix $\{T_{ik}\}$ and processing time matrix $\{p_{ik}\}$ are given as inputs. Besides, the following GA parameters are also given: population size N, crossover rate R_c and mutation rate R_m.
Step 1:	A random initial population $P(t=0)$ of size N is constructed in which each individual is generated using the GT algorithm with randomly selecting operations. The makespan of each individual is automatically calculated as an output of the GT algorithm.
Step 2:	Select randomly $N \times R_c$ individuals from $P(t)$ and pair them randomly. Apply the GT crossover (with built-in mutation of probability R_m) to each pair and generate new $N \times R_c$ individuals that are inserted into $P'(t)$. The rest of $P(t)$ members are just copied to $P'(t)$. As a result of the GT crossover, the makespan of each individual automatically calculated.
Step 3:	If the best makespan in $P'(t)$ is not as good as that in $P(t)$, then the worst individual in $P'(t)$ is replaced by the best individual in $P(t)$ (elitist strategy).
Step 4:	Reproduce $P(t+1)$ from $P'(t)$ by using the roulette wheel selection, in which each individual in $P'(t)$ is sorted in the descending order of its makespan so that the worst individual is numbered as x_1 and the best as x_N. Then the roulette wheel selection is applied with the fitness f of an individual x_i defined as $f(x_i) = i$ to obtain $P(t+1)$.
Step 5:	Set $t \leftarrow t+1$.
Step 6:	Repeat Steps 2 to 5 until some termination condition is met.
Step 7:	Output the best individual in $P(t)$ as the obtained best solution.

The GT crossover used in above algorithm is differing from other crossover because it operates directly on phenotype (a potential solution of the problem). The offspring generated from this crossover will represent active schedules, so there is no repairing process required.

3 A Modified Giffler and Thompson Genetic Algorithm

A simple modification to the GT-GA is made in order to achieve better-than or at-least-as-good-as solution in terms of speed of convergence comparing to the existing GT-GA with less computational time. It can be an alternative algorithm to use for the GT-GA. In the modified GT-GA, the pairing method will be different from the original GT-GA. There are several kinds of pairing method in GA such as pairing from top to bottom, random pairing, weighted random pairing and tournament selection. Each of the parent selection schemes results in a different set of parents. As such, the composition of the next generation is different for each selection scheme. Sometimes one works better than the other. Thus it is very difficult to give advice on which selection scheme works best. Tournament selection has been chosen in this research as the pairing method in the modified GT-GA while the existing GT-GA uses random pairing.

Tournament selection is another approach that closely mimics mating competition in nature which is to randomly pick a small subset of chromosomes from the mating pool, and the fittest chromosome in this subset becomes a parent [4]. The tournament repeats for every parent needed. Consider having six tournaments between two randomly selected chromosomes from the mating pool and selecting the fittest chromosomes from the tournament as parents. Results are shown in table 1. Thresholding and tournament selection make a nice pair, because the population never needs to be sorted. Using tournament selection chromosome 1 is paired with chromosome 2, chromosome 6 is paired with chromosome 3, and chromosome 4 is paired with chromosome 1.

Table 1: Generation of two random integers between 1 and 6

Tournament	Randomly selected chromosomes	Parent (fittest chromosomes)
1	3, 1	1
2	2, 5	2
3	6, 6	6
4	3, 4	3
5	4, 5	4
6	1, 1	1

Besides the above modification, step 3 of the algorithm 1 will be repeated between step 4 and step 5. This is because it is better to use the elitist strategy after the roulette wheel selection since roulette wheel selection does not guarantee the selection of the fittest individual. By adding this step after the roulette wheel selection, it is hoped that the resulting solution will achieve shorter computational time. The following is the modified algorithm for GT-GA.

Algorithm 2 A modified GA using the GT Crossover

Note:	The technological sequence matrix $\{T_{ik}\}$ and processing time matrix $\{p_{ik}\}$ are given as inputs. The following GA parameters are also given: population size N, crossover rate R_c and mutation rate R_m.
Step 1:	A random initial population $P(t=0)$ of size N is constructed in which each individual is generated using the GT algorithm with randomly selecting operations. The makespan of each individual is automatically calculated as an output of the GT algorithm.
Step 2:	Select randomly $N \times R_c$ individuals from $P(t)$ and pair them using tournament selection. Apply the GT crossover (with built-in mutation of probability R_m) to each pair and generate new $N \times R_c$ individuals that are inserted into $P'(t)$. The rest of $P(t)$ members are just copied to $P'(t)$. As a result of the GT crossover, the makespan of each individual automatically calculated.
Step 3:	If the best makespan in $P'(t)$ is not as good as that in $P(t)$, then the worst individual in $P'(t)$ is replaced by the best individual in $P(t)$ (elitist strategy).
Step 4:	Reproduce $P(t+1)$ from $P'(t)$ by using the roulette wheel selection, in which each individual in $P'(t)$ is sorted in the descending order of its makespan so that the worst individual is numbered as x_1 and the best as x_N . Then the roulette wheel selection is applied with the fitness f of an individual x_i defined as $f(x_i) = i$ to obtain $P(t+1)$.
Step 5:	If the best makespan in $P(t+1)$ is not as good as that in $P'(t)$, then the worst individual in $P(t+1)$ is replaced by the best individual in $P'(t)$ (elitist strategy).
Step 6:	Set $t \leftarrow t+1$.
Step 7:	Repeat Steps 2 to 5 until some termination condition is met.
Step 8:	Output the best individual in $P(t)$ as the obtained best solution.

From the algorithm, once the initial population is constructed, we will randomly select $N \times R_c$ individual from the population and pair them using Tournament Selection. The selected pairs are required to undergo crossover and mutation subsequently. Since our intention is to keep the individual with best makespan so that we might get the minimum makespan in the end of the process, we replace the worst individual with the best one. We call it as elitist strategy. This progress is followed by the roulette wheel selection as one of the most common reproduction operator. Again, we will apply elitist strategy after the reproduction for the same reason being.

4 Analysis

Two kinds of analysis are carried out in this research. First is the comparison for both the existing and modified programs. This comparison is to show whether the modified

program is more effective than the existing one. If it can achieve as-good-as or better-than result, it can be another alternative to solve JSSP instead of the existing GT-GA. The analysis is carried out by the visual prolog programming language. Since GT-GA is an algorithm based on blind search, best results will only be attained if there are more populations and generations for each run. However, when the sizes of the populations and generations increase, the processing is greatly affected by processor speed and internal memory capability. One of the reasons is that we need to find longest path in our algorithm. It is a need for the program to explore every possibility before we get the longest path. When the problem becomes bigger, the large size of population and generations may cause the program to explore those possibilities more frequently. Hence, it may require quite long time to solve even small benchmark problems with large population and generations, for example we need to use around 2 hours to solve FT10 with 10 generations and 100 populations (Intel 3 running at 1 GHz). Let's imagine a big tree with uncounted branches to have an idea of how the program works. Hence we have set it to be equal to 5 generations for each benchmark problems with the number of populations stated for each problem. The outcomes for each benchmark problems are thus only near optimal but not optimal although it is proved to have optimal solution when solved by Yamada's existing GT-GA [5]. However, the success achieved by Yamada is limited in the sense that the optimal makespan (the duration in which all operations for all jobs are completed) for the FT 10 problem is obtained only occasionally among many trials (4 times among 600 trials in 200 generations). That means for each trial, although the process has been repeated for 200 generations, we may not get the optimal solution. Furthermore, even in the Yamada's existing GT-GA, optimal solution cannot be achieved for FT 20 problem. However, considering the simplicity of the algorithm, the results are still interesting.

In the second analysis, some GA parameters such as population size, crossover rate and mutation rate are tested with different settings on the selected benchmark problem. The purpose of this analysis is to investigate the effect of different parameter settings. Hence 3 settings for each parameter are conducted so that the result is on the whole. We will still employ 5 generations for each parameter testing.

4.1 Comparison Using Benchmark Problems

To find the comparative merits of both the existing and modified GT-GA, they need to be tested on the same problems. Hence we can use the so called "benchmark problems" which provide a common standard on which all deterministic JSSP can be tested and compared. Thus it will be easy to gauge the strength and power of our algorithms from a statement such as "Algorithm P achieved a makespan of x in time y on benchmark problem Q." As the benchmark problems are of different dimensions and grades of difficulty, it is simple to determine the capabilities and limitations of a given method by testing it on the benchmark problems. Also the test findings may suggest the improvements required and where they should be made. The benchmark problems which have received the greatest analysis are the instances generated by Fisher and Thompson (FT) in 1963 (refer to OR Library site: http/www.ms.ic.ac.uk/jeb/pub/jobshop1.txt and jobshop2.txt). We have chosen the well-established FT 06 (6 jobs and 6 machines) and FT 10 (10 jobs and 10 machines) for the test (refer to appendix A and B for these instances). Both, the existing and the modified GT-GA are applied to these benchmark problems to explore their efficiencies and limitations. The GA parameters are set to be the same for every instance since the experiments are carried

out to compare the effectiveness for both algorithms. These parameters are set according to Yamada's thesis with some adjustment, which are

1) population size: 100,
2) crossover rate: 0.9,
3) mutation rate: 0.001.

(a) FT 06

For this benchmark problem, the comparison (in ten trials) for both the existing and modified GT-GA is shown in Table 2.

Table 2: Comparison for existing and modified GT-GA (FT 06)

	Makespan (processing time)	
Trial	Existing GT-GA (seconds)	Modified GT-GA (seconds)
1	79 (103.39)	62 (104.062)
2	65 (90.578)	71 (72.484)
3	71 (87.735)	66 (77.859)
4	60 (104.093)	76 (84.438)
5	83 (95.359)	77 (111.000)
6	78 (102.328)	71 (94.656)
7	77 (88.594)	69 (88.078)
8	76 (81.468)	74 (93.187)
9	72 (100.797)	70 (97.860)
10	60 (110.344)	73 (111.250)
Average	72.1 (96.469)	70.9 (93.487)

From Table 2, the integer values for average makespan are 72 and 71 and the processing time for both the existing and modified programs are 96.469 and 93.487. We can observe that the average performance for the modified GT-GA is a bit better than the existing GT-GA in either the average makespan or the average processing time. However, we still need to observe the performance of GT-GA on other benchmark problem before we can make any conclusion here. Figures 1 and 2 also show us that the performance of the modified GT-GA is as good as the existing GT-GA. For the reader's information, the optimal solution for this benchmark problem is 55 (refer to the OR Library site: http/www.ms.ic.ac.uk/jeb/pub/jobshop1.txt and jobshop2.txt).

(b) FT 10

For this benchmark problem, the comparison (in ten trials) for both the existing and modified GT-GA is as follows:

From table 3 the integer values for the average makespan are 1387.5 and 1375.8 and the processing time for both the existing and modified programs are 3915.922 and 3726.569. We can also observe that the average performance for the modified GT-GA is a bit better

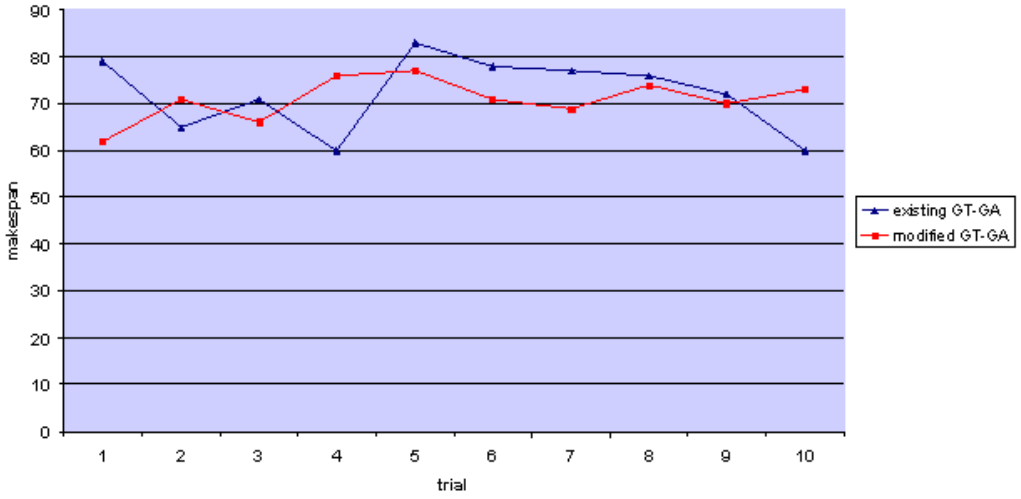

Figure 1: Graph for comparison between the existing and modified GT-GA in makespan objective (FT 06)

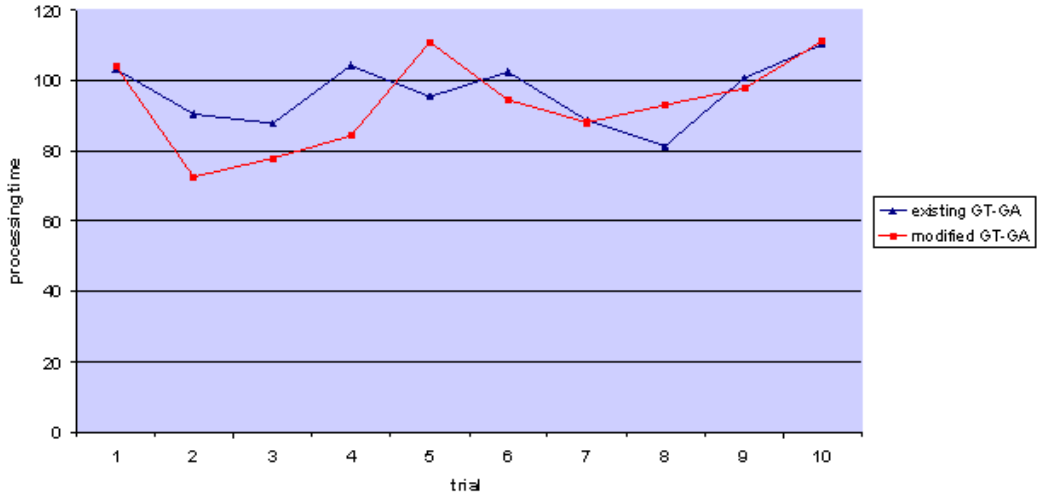

Figure 2: Graph for comparison between the existing and modified GT-GA in processing time (FT 06)

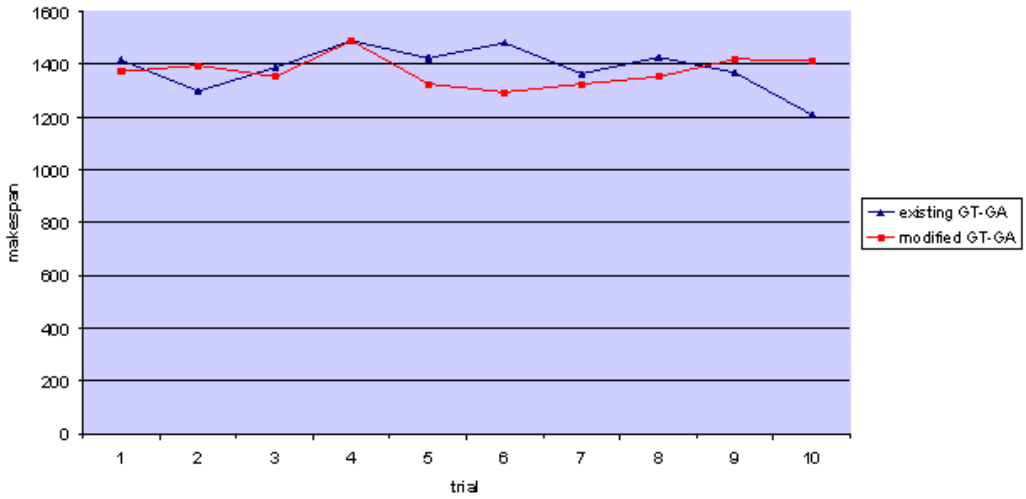

Figure 3: Graph for comparison between the existing and modified GT-GA in makespan Objective (FT 10)

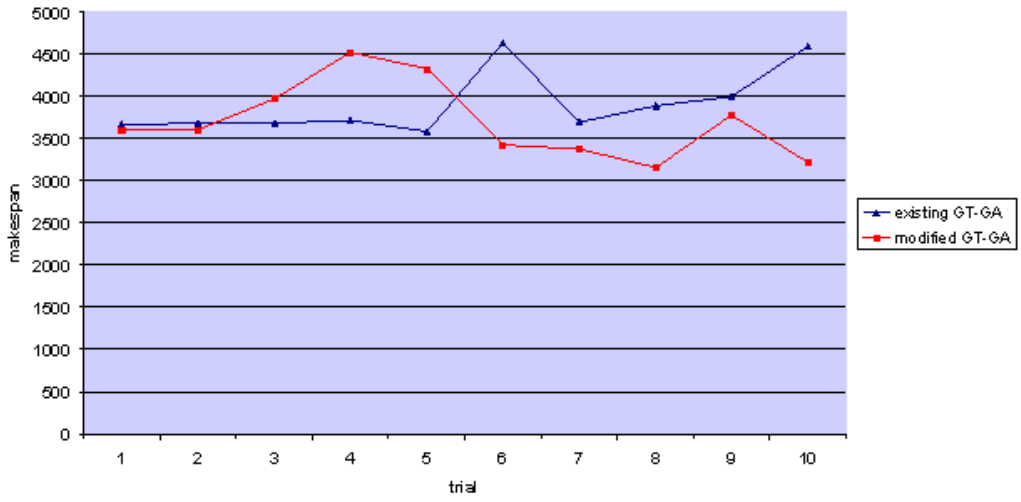

Figure 4: Graph for comparison between the existing and modified GT-GA in processing time (FT 10)

Table 3: Comparison for existing and modified GT-GA (FT 10)

	Makespan (processing time)	
Trial	Existing GT-GA (seconds)	Modified GT-GA (seconds)
1	1416 (3672.515)	1375 (3610.078)
2	1299 (3683.172)	1396 (3610.078)
3	1390 (3688.297)	1354 (3974.500)
4	1492 (3714.718)	1490 (4521.063)
5	1423 (3585.734)	1328 (4327.391)
6	1483 (4638.875)	1297 (3431.313)
7	1366 (3690.391)	1325 (3381.562)
8	1429 (3893.797)	1356 (3153.375)
9	1369 (3999.829)	1421 (3779.437)
10	1208 (4591.890)	1416 (3223.569)
Average	1387.5 (3915.922)	1375.8 (3726.569)

than the existing GT-GA in either the average makespan or the average processing time. Figures 3 and 4 also show us that the performance of the modified GT-GA is as good as the existing GT-GA. For the reader's information, the optimal solution for this benchmark problem is 930.

From both the benchmark problems shown above, we can see that the average performance for the modified GT-GA is encouraging. However, the result shown is not arbitrary since GT-GA is a process starting with a set of random initial solutions. Thus, any of the GT-GA might perform better than the other one if their initial solutions are good. As a conclusion, we can say that the modified GT-GA can perform as well as the existing GT-GA in less computational time.

4.2 Effect of the GA Parameters

GA parameters play an important role in observing the effectiveness of our algorithms. Different settings for each GA parameters produce different effects as applied on different instances. The modified GT-GA is used to test with the different parameters in this section. FT 06 will be used as our benchmark problem in this section.

(a) Population Size
In this section, we will test the effect of different population sizes, such as population size 1, 100 and 500. The other GA parameters are as the following which is according to Yamada's thesis [5]:

1) crossover rate: 0.9,
2) mutation rate: 0.001.

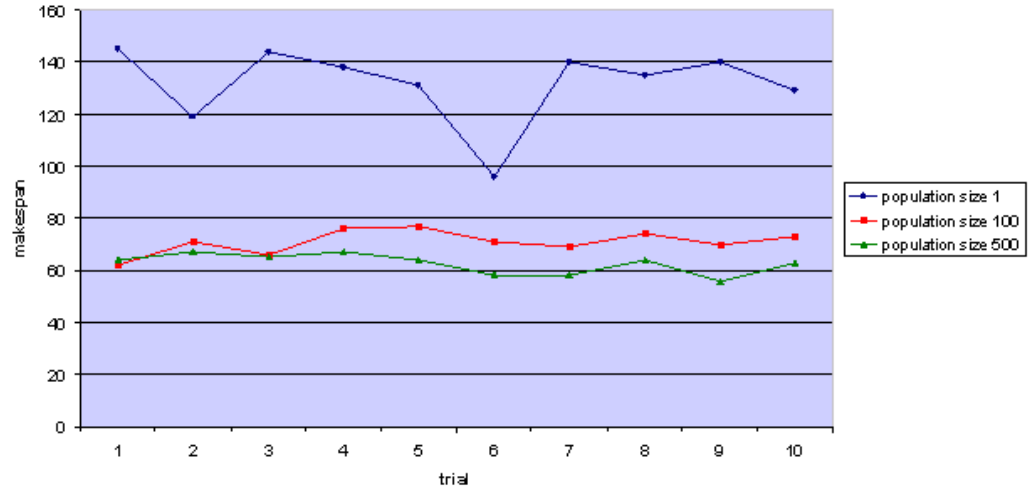

Figure 5: Graph for comparison of population size 1, 100 and 500 in makespan objective

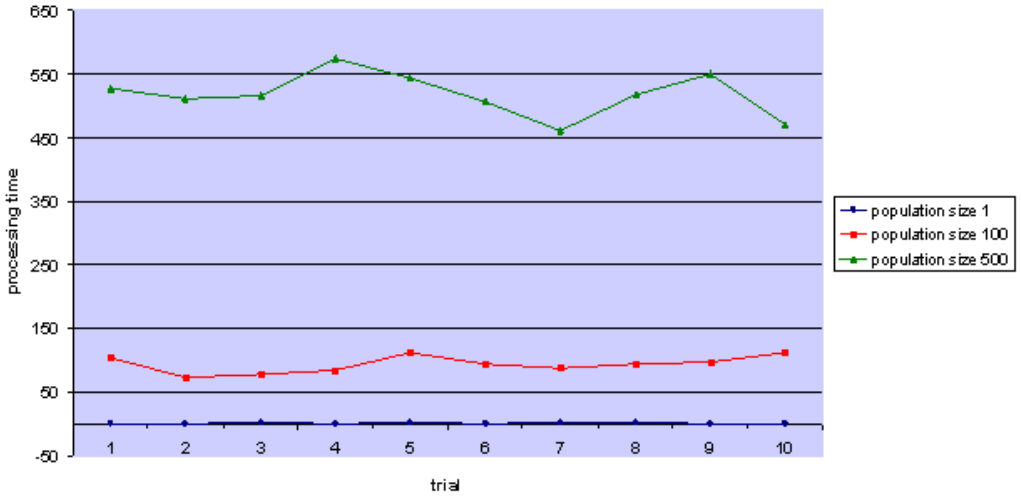

Figure 6: Graph for comparison of population size 1, 100 and 500 in processing time

Table 4: Minimum makespan for different population size

Trial	Makespan for population size 1 (processing time, seconds)	Makespan for population size 100 (processing time, seconds)	Makespan for population size 500 (processing time, seconds)
1	145 (0.906)	62 (104.062)	64 (527.031)
2	119 (0.625)	71 (72.484)	67 (511.344)
3	144 (1.922)	66 (77.859)	65 (516.344)
4	138 (0.719)	76 (84.438)	67 (573.672)
5	131 (1.156)	77 (111.000)	64 (542.828)
6	96 (0.375)	71 (94.656)	58 (506.860)
7	140 (1.000)	69 (88.078)	58 (460.968)
8	135 (1.328)	74 (93.187)	64 (517.250)
9	140 (0.860)	70 (97.860)	56 (549.672)
10	129 (0.781)	73 (111.250)	63 (470.610)
Average	131.7 (0.967)	70.9 (93.487)	62.6 (517.658)

From table 4, the integer values for average makespan are 132, 71 and 63. It is observed that smaller makespan can be achieved with the increasing number of population size. However, the processing time needed is much longer for big population size; for example, the average time needed for population size 500 is 500 times longer than population size 1. We can also observe this from figures 5 and 6. Hence, it is advisable to take bigger population sizes for bigger problems so that the convergence to minimum makespan will be faster, and smaller population size for smaller problem so that we will not waste time finding minimum makespan while we can find it faster with small population sizes.

(b) Crossover Rate

In this section, we will test on the effect of different crossover rates, namely 0.1, 0.5 and 0.9. The other GA parameters are, according to Yamada's thesis [5], as follows:

1) population size: 100,

2) mutation rate: 0.001.

From table 5, the integer values for average makespan are 78, 73 and 71. Since the differences between makespan for crossover rate at 0.5 and 0.9 are not significant, we can use crossover rate in the range from 0.5 to 0.9 which will produce better makespan than crossover rate 0.1. Although crossover rate 0.1 needs shorter time to produce the solutions, we prefer the crossover rate from 0.5 to 0.9 because we want to have more individual solutions to undergo the crossover so that we might get better solutions in less generations. We can also observe this from figures 7 and 8. This is about the same as Coley's work[3] which stated that typical values for crossover rate are 0.4 to 0.9.

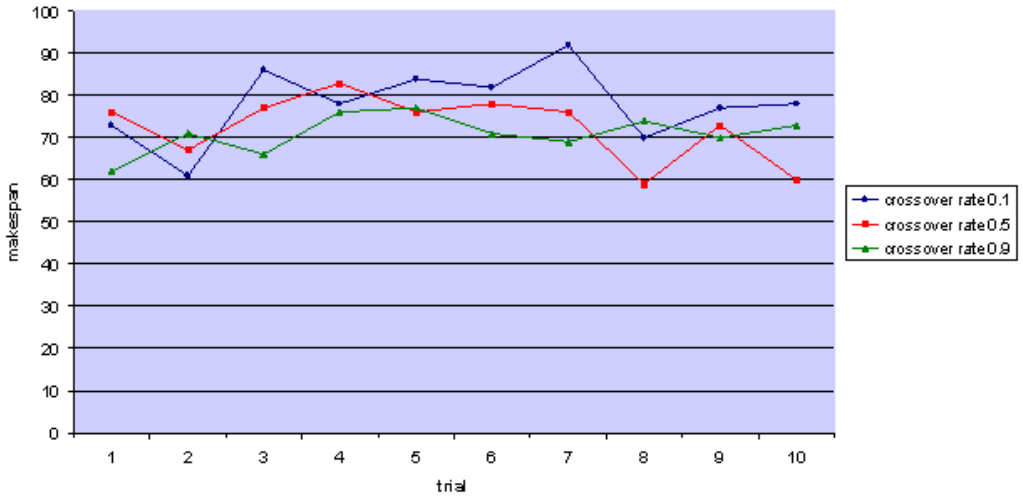

Figure 7: Graph for comparison of crossover rates 0.1, 0.5 and 0.9 in the makespan objective

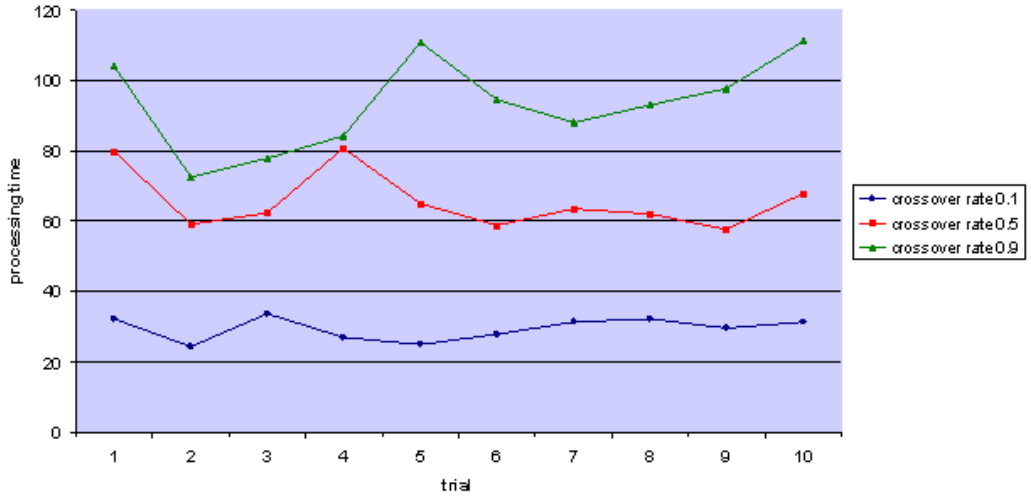

Figure 8: Graph for comparison of crossover rates 0.1, 0.5 and 0.9 in terms of processing time

Table 5: Minimum makespan for different crossover rate

Trial	Makespan for crossover rate 0.1 (processing time, seconds)	Makespan for crossover rate 0.5 (processing time, seconds)	Makespan for crossover rate 0.9 (processing time, seconds)
1	73 (32.125)	76 (79.921)	62 (104.062)
2	61 (24.344)	67 (59.266)	71 (72.484)
3	86 (33.625)	77 (62.469)	66 (77.859)
4	78 (26.906)	83 (80.687)	76 (84.438)
5	84 (24.844)	76 (64.985)	77 (111.000)
6	82 (27.859)	78 (58.953)	71 (94.656)
7	92 (31.281)	76 (63.562)	69 (88.078)
8	70 (32.187)	59 (62.187)	74 (93.187)
9	77 (29.547)	73 (57.750)	70 (97.860)
10	78 (31.344)	60 (68.015)	73 (111.250)
Average	78.1 (29.406)	72.5 (65.780)	70.9 (93.487)

(c) Mutation Rate

In this section, we will test on the effect of different mutation rates, such as 0.001, 0.01 and 0.1. The other GA parameters are, according to Yamada's thesis [5], as follows:

1) population size: 100,

2) crossover rate: 0.9.

From table 6, the integer values for average makespan are 71, 68 and 69. Although we can get better result with high mutation rate, since mutation is the second way a GA explore a solution space, it is advisable to use lower mutation rate in order to keep some traits in the parent solutions. In addition, too high a mutation rate will incur too much perturbation where the offspring will start losing their resemblance to their parent. Hence the algorithm will lose the ability to learn from the history search. We can see this by observing the average makespan we get for mutation rate 0.1 is bigger than average makespan for mutation rate 0.01. We can also observe that the range for the minimum makespan (59, in second trial) and maximum makespan (79, in first trial) for mutation rate 0.1 is 20 which indicates that the algorithm has the tendency to change to random search. In addition, the processing time needed is shorter for smaller mutation rate. We can also observe this from figures 9 and 10.

In conclusion, we should balance between these parameter settings in order to obtain the intended solutions.

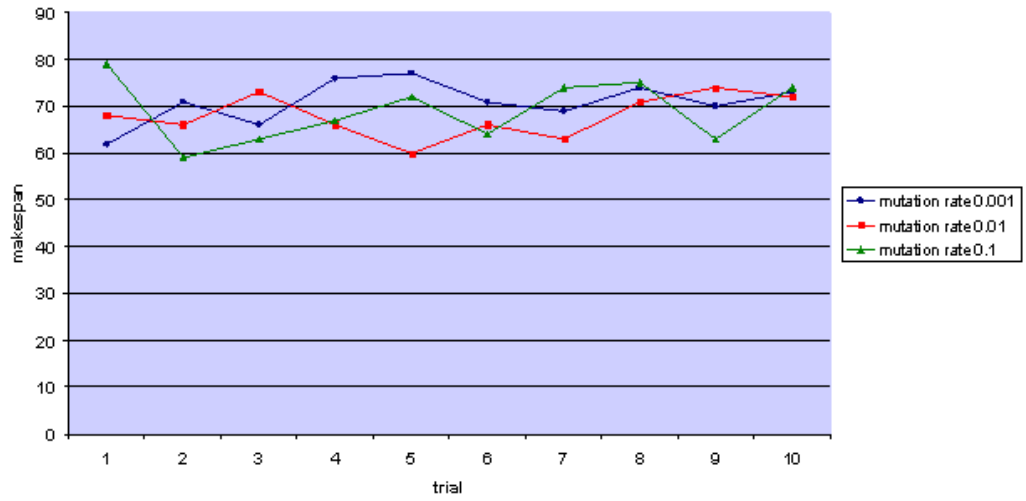

Figure 9: Graph for comparison for mutation rates 0.001, 0.01 and 0.1 in makespan objective

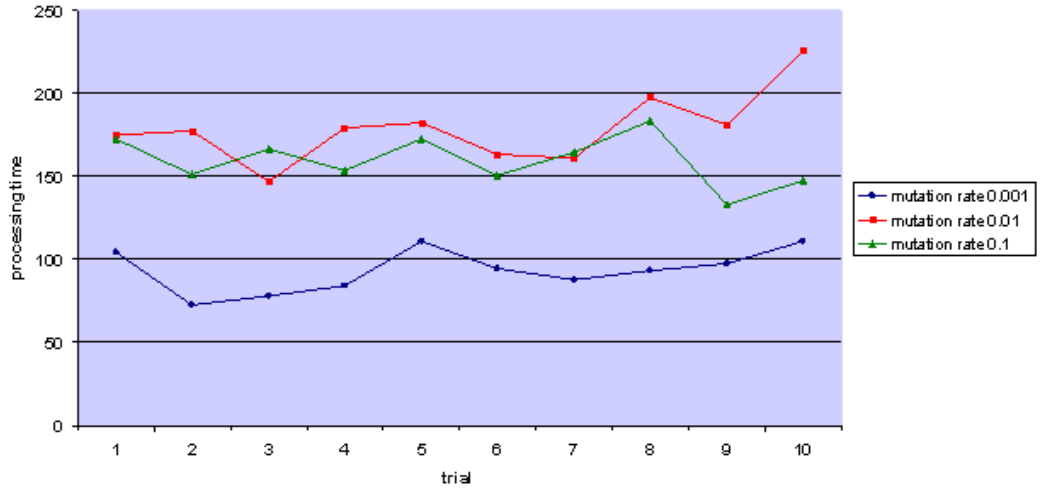

Figure 10: Graph for comparison for mutation rates 0.001, 0.01 and 0.1 in processing time

Table 6: Minimum makespan for different mutation rate

Trial	Makespan for mutation rate 0.001 (processing time, seconds)	Makespan for mutation rate 0.01 (processing time, seconds)	Makespan for mutation rate 0.1 (processing time, seconds)
1	62 (104.062)	68 (175.016)	79 (172.469)
2	71 (72.484)	66 (177.281)	59 (151.203)
3	66 (77.859)	73 (147.140)	63 (166.281)
4	76 (84.438)	66 (179.188)	67 (153.938)
5	77 (111.000)	60 (182.422)	72 (172.344)
6	71 (94.656)	66 (163.219)	64 (150.313)
7	69 (88.078)	63 (160.875)	74 (164.578)
8	74 (93.187)	71 (197.609)	75 (183.750)
9	70 (97.860)	74 (181.359)	63 (132.969)
10	73 (111.250)	72 (225.516)	74 (147.656)
Average	70.9 (93.487)	67.9 (178.963)	69.0 (159.550)

5 Conclusion

GT-GA is chosen as our method to solve JSSP in this research because each solution is always a feasible and active schedule. Hence there is no repairing process required which can increase the efficiency of GA approach. The wasteful intermediate decoding steps can also be skipped since the crossover operates directly on phenotype. This algorithm is modified as a new approach to solve JSSP. From the comparison experiment, we can conclude that the modified GT-GA is as well as the existing GT-GA and hence the modified GT-GA can be chosen as one of our alternative to solve JSSP other than the existing GT-GA since it can also produce the solutions in less computational time. The algorithm is also tested with different parameter sets, such as population size, crossover rate and mutation rate. It is observed that each of these parameter setting have great influence to the GA. Solutions which are not near optimal or wasting a lot of time might be due to the unsuitable setting of the parameters, such as high crossover rate with high mutation rate or its opposite and low population size with large problem or its opposite. Hence we should take care of these kinds of setting before we start to solve JSSP with GA.

Acknowledgement

The authors would like to thank Universiti Teknologi Malaysia (UTM) for providing facilities and financial support (Vote 75119) for this work.

References

[1] S.Salhi, *Heuristic Search Methods*, University of Birmingham Press, Birmingham, England, 147-175, 1998.

[2] D. E. Goldberg, *Genetic Algorithm in Search, Optimization, and Machine Learning*, Addison-Wesley, Canada, 1-25, 1989.

[3] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific, Singapore, 1-117, 1999.

[4] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, John Wiley & Sons, Canada, 25-48, 1998.

[5] T. Yamada, *Studies on Metaheuristics for Jobshop and Flowshop Scheduling Problems*, Kyoto University: Ph. D. thesis, 2003.

[6] R. W. Cheng, M. S. Gen, and Y. Tsujimura, *A Tutorial Survey of Job-Shop Scheduling Problems Using Genetic Algorithms, part II: Hybrid Genetic Search Strategies*, Elsevier. 36, 343-364, 1996.

Appendix A

Job	Machine (processing time)					
1	3 (1)	1 (3)	2 (6)	4 (7)	6 (3)	5 (6)
2	2 (8)	3 (5)	5 (10)	6 (10)	1 (10)	4 (4)
3	3 (5)	4 (4)	6 (8)	1 (9)	2 (1)	5 (7)
4	2 (5)	1 (5)	3 (5)	4 (3)	5 (8)	6 (9)
5	3 (9)	2 (3)	5 (5)	6 (4)	1 (3)	4 (1)
6	2 (3)	4 (3)	6 (9)	1 (10)	5 (4)	3 (1)

Appendix B

Job	Machine (processing time)									
1	1 (29)	2 (78)	3 (9)	4 (36)	5 (49)	6 (11)	7 (62)	8 (56)	9 (44)	10 (21)
2	1 (43)	3 (90)	5 (75)	10 (11)	4 (69)	2 (28)	7 (46)	6 (46)	8 (72)	9 (30)
3	2 (91)	1 (85)	4 (39)	3 (74)	9 (90)	6 (10)	8 (12)	7 (89)	10 (45)	5 (33)
4	2 (81)	3 (95)	1 (71)	5 (99)	7 (9)	9 (52)	8 (85)	4 (98)	10 (22)	6 (43)
5	3 (14)	1 (6)	2 (22)	6 (61)	4 (26)	5 (69)	9 (21)	8 (49)	10 (72)	7 (53)
6	3 (84)	2 (2)	6 (52)	4 (95)	9 (48)	10 (72)	1 (47)	7 (65)	5 (6)	8 (25)
7	2 (46)	1 (37)	4 (61)	3 (13)	7 (32)	6 (21)	10 (32)	9 (89)	8 (30)	5 (55)
8	3 (31)	1 (86)	2 (46)	6 (74)	5 (32)	7 (88)	9 (19)	10 (48)	8 (36)	4 (79)
9	1 (76)	2 (69)	4 (76)	6 (51)	3 (85)	10 (11)	7 (40)	8 (89)	5 (26)	9 (74)
10	2 (85)	1 (13)	3 (61)	7 (7)	9 (64)	10 (76)	6 (47)	4 (52)	5 (90)	8 (45)