Chapter 4

Multi-Agent based Genetic Algorithm for JSSP

4.1 Introduction

Genetic algorithm (GA) is a stochastic search and an optimization technique which mimics the biological evolution and is based on the Darwinian principle of ‘survival of the fittest’. The principle of survival of the fittest emphasises on good solutions and eliminates the bad ones. In terms of searching behaviour, the GA falls into the category of global optimisation methods. The GA differs from other conventional optimization algorithms in that the GA operates on a search space of a population of solutions and evolves a set of the solutions rather than a single solution. The GA was first conceived by Professor John Holland of University of Michigan in 1965 (Holland, 1975). Later on, many researchers became interested in the GA. The first approach to solve the JSSP using the GA was introduced by Davis (1985). Since then, many papers on solving the JSSP through the GA have appeared. Yamada and Nakano (1992) proposed new representation of chromosome for the JSSP.
An in-depth survey of the application of the GA in the JSSP is found in (Cheng et al., 1996). To make the GA as an efficient optimization technique, a famous principle of hybridization is 'hybridize where possible' was proposed by Davis (1991). Since then, many researchers have integrated the GA with other heuristic techniques and got more efficient results. One hybrid GA was given by Wang and Zheng (2002), where a Modified Genetic Algorithm (MGA) has been proposed. In MGA, the classical mutation operator has been replaced with the metropolis sample process of simulated annealing. This enhanced the neighborhood search and prevented from premature convergence. In recent years, the concept of the Multi-Agent System (MAS) has also been employed with the GA to solve a JSSP and other scheduling problems. Cardon et al. (2000) proposed a multi-agent system with a GA to solve a JSSP. Their proposed model contains distributor agent that distributes the jobs among machine agents. Also, Chen et al. (2004) proposed a MAS based on a GA to solve a JSSP, which accelerates the GA in intelligent way. The proposed MAS model contains one management agent and number of execute agents, where each machine has one execute agent. (Zhong et al., 2005) developed multi-agent evolutionary algorithm for JSSP, named MAEA_JSP, where the agents live in lattice environment (i.e. each agent has direct connection with only four neighbors). Each agent in MAEA_JSP algorithm has competitive behaviour with its neighbours and self-learning behaviour that uses local search technique.

In this chapter, basic concepts of the GA are described. The chapter includes the proposed multi-agent GA model for the JSSP which called MAGA and its implementation. The experiments conducted on some of standard benchmarks of the JSSP and the results are compared with the results of the research paper (Zhong et al., 2005).
4.2 Concepts of the Genetic Algorithm

The GA works with a population of candidate solutions, so it carries out a multiple search simultaneously. A solution is usually encoded as a sequence of symbols which is called a chromosome or an individual. Each chromosome has a fitness measure of adaptation called a fitness value, which is often related to the objective function and calculated using a proposed fitness function. Thus, the quality of solution is evaluated by a fitness function. The fitness function comprises the objective and constraint violations. A fitness value of a solution affects the probability that this solution is chosen in the selection process to participate in subsequent reproductive operations. The GA starts from an initial population and selects some “parent” solutions to generate new “child” solutions (offspring) by applying genetic operators (crossover and mutation). While a crossover operator usually mixes subsequences of two selected parent chromosomes, a mutation operator perturbs chromosomes. The created child solutions are added to the population and the population is reduced to its original size by removing some solutions according to their fitness values. Afterwards, the same process is applied to the new population. The simple GA is given as follows:

**Simple GA**

1. Generate an initial population POP;
2. Calculate the fitness value for each individual of POP;
3. While the stopping criterion has not satisfied
4. Choose parent solutions from current population for mating;
5. Create offspring from the chosen parents by crossover with a certain probability (P_c);
6. Mutate the offspring with a certain probability (P_m);
7. Calculate the fitness values of the offspring;
8. Add the offspring to the new POP;
9. Repeat from the step 3.

The population of the chromosomes evolves iteratively until satisfying the stopping criterion, Figure 4.1 shows the basic cycle of the GA.

![Figure 4.1 Basic cycle of the genetic algorithm](image)

The major reasons for popularity of the GA in various search and optimization problems are the global perspective, widespread applicability, and inherent parallelism. The GA is a powerful optimization tool that satisfies the following features:

- **Stochastic**: GA is a stochastic algorithm; randomness has an essential role in genetic algorithms. Selection and genetic operations needs random procedures.

- **Parallelism**: GA operates on a population of solutions. The algorithm can recombine different solutions to get better ones and so, it can use the benefits of assortment. A population based algorithm is also very amenable for parallelization.
• **Robustness:** The robustness of the algorithm should also be mentioned as an essential feature for the success of the algorithm. Robustness refers to the ability to perform consistently well on a broad range of problem types. There is no particular requirement on the problem before using a GA, so it can be applied to resolve any problem.

Like most stochastic methods, the GA is not guaranteed to find the global optimum solution to a problem, it satisfied with finding ‘acceptably good’ solutions to the problem.

### 4.3 GA Parameters

In order to implement a GA, a number of parameters are required to be predefined; i.e., the size of the population, the selection method, the crossover probability, the mutation probability, and the termination criterion.

#### 4.3.1 Population Size

For each problem, the population size depends on the complexity of the problem. Most often the initial population is randomly initialized. Sometimes, a kind of heuristic may also be used to seed the initial population. Thus, the mean fitness of the population is expected to be high and it may help the GA to converge to a set of good solutions faster. If the population lacks diversity, the algorithm explores a small part of the search space and does not find global optimal solutions. A large population is quite useful. But it requires much more computational cost, memory and time.
4.3.2 Selection Methods

A selection method picks chromosomes from the population according to their fitness values. The higher the fitness value, the more chance the individual may be selected. The selection process drives the GA to improve the fitness of the population over the successive generations. A selection pressure is defined as the degree to which the better individuals are favoured. The ideal selection strategy should be such that it is able to adjust its selection pressure and population diversity so as to fine-tune GA search performance. The convergence rate of GA is largely determined by the magnitude of the selection pressure, with a higher selection pressure resulting in a higher convergence rate. Selection methods should also preserve population diversity, as this helps to avoid premature convergence. Some of the selection methods are discussed as follows:

- **Roulette wheel selection:** The roulette wheel selection is one of the traditional GA selection techniques and the commonly used proportionate reproductive method, where a chromosome is selected from the population with a probability proportional to its fitness value. The principle of roulette selection is a linear search through a roulette wheel with the slots in the wheel weighted in proportion to the fitness values of the individuals. The Roulette process is as follows: The fitness value of an individual is divided by the total fitness of the population. Each individual of the population is assigned a slice of the roulette wheel, the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the
pool of parents for the next generation. As a result, fit individuals are not guaranteed to be selected, but somewhat have a greater chance. Figure 4.2 shows a roulette wheel with five individuals and their selection probabilities.

![Figure 4.2 Roulette wheel selection](image)

The roulette wheel selection process is summarised as follows:

1. Sum the fitness of all the population members, $f_{\text{sum}}$.
2. Choose a random number, $R$, between 0 and $f_{\text{sum}}$.
3. Add together the fitness of the population members (one at a time) until the sum is greater than $R$. The last individual added is the selected individual.

This selection method encounters a problem when the fitness values differ very much. If the fitness of the best individual occupies 90% of the roulette wheel, then other individuals have too few chances to be selected.

- **Rank selection**: The rank selection ranks the population and every chromosome receives fitness from the ranking. The worst has fitness 1 and the best has fitness $N$, where $N$ is the number of individuals in the population. Rank Selection results in slow convergence but
prevents too quick convergence. It also keeps up selection pressure when the fitness variance is low. It preserves diversity and hence leads to a successful search. In effect, potential parents are selected and the tournament is held to decide which of the individuals will be the parent. This can be achieved in many ways. One way is by selecting a pair of individuals at random and generating a random number $R \in [0,1]$. The value of $r$ is a parameter to the ranking method. If $R < r$, use the first individual as a parent. If $R \geq r$, then use the second individual as the parent. This process is repeated to select the second parent. Another way is by selecting two individuals at random. The individual with the highest evaluation becomes the parent.

- **Tournament selection:** The tournament selection strategy provides selective pressure by holding a tournament competition among $K$ individuals, where $K$ is less than the population size. The selected individual from the tournament is the one with the best fitness. The tournament is repeated to select other individuals. As result, the chosen individuals have fitness values which are higher than the average fitness values of the population.

### 4.3.3 Crossover Probability

A crossover probability $P_c$ is a parameter to describe how often the crossover would be performed. If there is no crossover, offspring are exact copies of parents. In the crossover operation, offspring are made from parts of both parents. If the crossover probability is 100%, then all the offspring are made by the crossover operation. The crossover operation aims that the new chromosomes would contain good parts of the old chromosomes and
therefore some of the new chromosomes would be better. However, it is good to leave some part of the old population survive to the next generation.

4.3.4 Mutation Probability

A mutation probability $P_m$ decides how often parts of offspring need to be mutated. If there is no mutation, offspring are directly copied after crossover without any change. If the mutation operation is performed, one or more parts of the offspring are changed. If the mutation probability is 100%, the whole offspring is changed. If the mutation probability is 0%, nothing is changed. Mutation generally prevents the GA from falling into local optima. If the mutation probability is high, then the GA changes to a random search.

4.3.5 Termination Criterion

The execution of the GA is stopped when a prescribed maximum number of generations has been reached, when there is no change in the best individual of the population for specified number of generations, or when the optimum solution is found.

4.4 Variants of the Genetic Algorithm

Variants of GAs have been developed by researchers to improve the efficiency and robustness of the GA algorithm. Some of these variants are:
4.4.1 Elitist Genetic Algorithm

In elitist genetic algorithm (EGA), elitism strategy is added to the selection scheme where a specific number of the best individuals from the parent population are copied directly into the next generation. Elitism guarantees that the fittest individual in the next population will be better or equal than the fittest individual in the current population. Elitism can rapidly increase performance of the GA, because it prevents losing the best found solution. Elitism may lead to premature convergence if it is not used with caution.

4.4.2 Adaptive Genetic Algorithm

Adaptive Genetic Algorithm (AGA) is a genetic algorithm whose parameters such as the population size, the crossover probability, or the mutation probability are varied while the genetic algorithm is running (Srinivas M., and Patnaik L., 1994). A simple example is that the higher mutation rate is chosen when the population does not improve after a specific number of generations. Vice versa, the mutation rate is decreased as soon as an improvement of the population occurs.

4.4.3 Hybrid Genetic Algorithm

The hybridizing of GA with other techniques is made in order to improve the optimization ability of the algorithm (Davis, 1991). The role of local search in the context of the GA has received significant consideration. Because of the complementary properties of the GA and conventional heuristics, a hybrid approach often outperforms either of the methods operating alone. One of the most common forms of the hybrid GA is to incorporate local search techniques to the GA after generating new offspring by genetic operations. A local search performs quick and
localized optimization in order to improve the offspring before evaluating its fitness. With the hybrid approach, a GA is used to perform global exploration among the population, while heuristic methods are used to perform local exploitation around chromosomes. A hybrid GA with a local search may be given as follows:

**Hybrid GA**

Let the generation number \( G = 1 \);
Initialize the population \( \text{POP}(G) \);
Apply local search to improve the individuals;
Evaluate \( \text{POP}(G) \);
while (termination condition is not satisfied) do
begin
Select parents from \( \text{POP}(G) \);
Crossover the selected Parents;
Mutate the generated offspring;
Apply local search to improve offspring;
Evaluate the new offspring;
Copy offspring to the new population \( \text{POP}(G+1) \);
\( G \leftarrow G + 1 \);
end

### 4.5 Chromosome Representations for JSSP

In the GA, each chromosome represents a potential solution, in which the solution parameters are encoded and stored. Originally, a binary encoding (i.e. string of '0's and '1's) was adopted, but this representation is not suitable for all types of problems. Generally, an appropriate solution representation must be developed for the problem such that encoding and decoding of the chromosome string does not significantly increase the
computational overhead and the new offspring generated by the genetic operators are feasible or easily repairable. For many complex problems like JSSP, it is usually very difficult to represent a solution with the bit string encodings. Therefore, many non-binary string encodings have been suggested and many new genetic operators have been adapted to cope with these non-binary string encodings. Regarding JSSP, many chromosome representations have been proposed in literature (Cheng et al, 1996). Some of JSSP representations are:

4.5.1 Disjunctive Graph-based Representation

Tamaki and Nishikawa (1992) proposed a disjunctive graph-based representation (DGR), which also can be viewed as one kind of job pair relation-based representation. The A chromosome consists of a binary string corresponding to an order list of disjunctive arcs in the graph.

For example, consider the JSSP instance (3-job and 3-machine) which has been given in Table 4.1.

<table>
<thead>
<tr>
<th>Table 4.1 JSSP (3×3) instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jobs</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>J₁</td>
</tr>
<tr>
<td>J₂</td>
</tr>
<tr>
<td>J₃</td>
</tr>
</tbody>
</table>

The disjunctive graph for this instance is shown in Figure 4.3, where eᵢⱼ stands for the disjunctive arc between nodes i and j and it is defined as follows:
- $e_{ij} = 0$, settle the orientation of the disjunctive arc from the node $i$ to the node $j$.

- $e_{ij} = 1$, settle the orientation of the disjunctive arc from the node $j$ to the node $i$.

The nodes of the disjunctive graph in Figure 3.3 are numbered from 1 to 9 to facilitate the notations of the disjunctive arcs. Suppose the ordered list of disjunctive arcs is given as $e_{1,4}$, $e_{1,8}$, $e_{4,8}$, $e_{2,7}$, $e_{2,6}$, $e_{6,7}$, $e_{3,5}$, $e_{3,9}$, $e_{5,9}$. A chromosome $[0, 0, 1, 1, 0, 0, 0, 1, 1]$ represents the orientation of these disjunctive arcs. The JSSP is to settle the orientation of the disjunctive arcs such that the resulting graph is acyclic. The acyclic disjunctive graph guarantees that there are no precedence conflicts between operations. It is easy to notice that an arbitrary setting of a chromosome may yield a cyclic disjunctive graph, i.e., the schedule is infeasible due to violation of any of the precedence constraints. Therefore, the disjunctive graph-based chromosome representation is not used to represent a schedule but only used as a decision preference with a path-based procedure to deduce the schedule. During the process of deduction, when conflict of two nodes
(operations) occurs on one machine, the corresponding bit of the chromosome is used to settle the processing order of the two operations.

4.5.2 Job-based Representation

Job-based representation (JOR) has been used by Holsapple et al. (1993). This representation consists of a list of n jobs and a schedule is constructed according to the sequence of jobs. For a given sequence of jobs, all operations of the first job in the list are scheduled first, and then the operations of the second job in the list are considered. The first operation of the job under treatment is allocated in the earliest available time for the corresponding machine the operation requires, and then the second operation, and so on until all operations of the job are scheduled. The process is repeated with each of the jobs in the list considered in the appropriate sequence. Any permutation of jobs corresponds to a feasible schedule.

Consider the same JSSP instance in Table 4.1 and suppose a chromosome is given as [2, 3, 1]. The first job to be processed is job J₂. Firstly, the operations of job J₂ are scheduled. The precedence constraints for J₂ are \( o_{2,1} \rightarrow o_{2,3} \rightarrow o_{2,2} \). Then the job J₃ is processed. The precedence constraints for J₃ are \( o_{3,2} \rightarrow o_{3,1} \rightarrow o_{3,3} \). Each of its operations is scheduled in the best available processing time. Lastly, the operations of job J₁ are scheduled. The precedence constraints for J₁ are \( o_{1,1} \rightarrow o_{1,2} \rightarrow o_{1,3} \) as shown in Figure 4.4.
4.5.3 Operation-based Representation

The operation-based representation (OBR) has been proposed by Gen et al. (1994). This representation encodes a schedule as a sequence of operations and each gene stands for one operation. For a JSSP with n-job and m-machine, a chromosome contains n×m genes. Each job appears in the chromosome exactly m times and each gene does not indicate a specific operation of a job but refers to a unique operation which is context-dependent. It is easy to see that any permutation of the chromosome always yields a feasible schedule. A schedule is decoded from a chromosome by translating the chromosome to a list of ordered operations and then generating the schedule by a one-pass heuristic procedure based on this list. In the one-pass heuristic procedure, the first operation in the list is scheduled first, then the second operation, and so on. Each operation under treatment is allocated in the earliest available time for the required machine.

Consider the same JSSP instance in Table 4.1 and suppose a chromosome is given as [2, 1, 1, 1, 2, 2, 3, 3, 3]. The chromosome can be translated into a unique ordered list of operations of o_{2,1}→ o_{1,1}→ o_{1,2}→
Chapter 4. MAGA for JSSP

The resulting schedule is shown in Figure 4.5.

Figure 4.5 Gantt chart for OBR [2, 1, 1, 1, 2, 2, 3, 3, 3]

4.5.4 Machine-based Representation

Dorndorf and Pesch (1995) proposed a machine-based representation (MBR), where a chromosome is encoded as a sequence of machines and a schedule is constructed with the shifting bottleneck heuristic procedure (SBP) that has been described in Section 2.5.2.2. The difference between the SBP and the GA is that the bottleneck is no longer a decision criterion for the choice of the next machine, which is controlled by a given chromosome. Let M be a set of already sequenced machines and a chromosome is given as [M_1, M_2, ... M_m]. The procedure of deducing the schedule from the chromosome is as follows:

**Procedure DeduceSchedule_MBR**

1. Let M = [], i = 1, and the chromosome [M_1, M_2, ... M_m]
2. Sequence the machine M_i optimally
3. Update set M ← M ∪ {M_i}.
4. Reoptimize the sequence of each critical machine M_i ∈ M in turn, while keeping the other sequences fixed.
5. i ← i + 1.
6. if i ≤ m then repeat from the step 2, else stop.
4.6 Crossover for JSSP

Crossover is performed by exchanging chromosome genes between two selected parents. The selected parents are recombined to produce a new population of offspring. The crossover operation is governed by a crossover probability $P_c$. The objective of performing crossover is to obtain better chromosomes by exploiting partial information contained in two relatively good parents. The type of crossover operator depends on the chromosome representation scheme for a given problem and the problem constraints. Traditionally, many crossover operators have been proposed for the binary representation. These binary crossover operators are suitable for the DGR representation in section 4.5.1. Some of the binary crossover operators are described below:

4.6.1 Single-Point Crossover

In the single-point crossover, two mating chromosomes are cut once at corresponding points and the sections after the cuts exchanged. The crossover point is chosen randomly. Figure 4.6 illustrates single-point crossover and it can be observed that the bits next to the crossover point (dotted line) are exchanged to produce children.

![Figure 4.6 Single-point crossover](image)

Parent 1 1 0 1 1 0 1 0 1
Parent 2 1 0 1 0 1 1 1

Child 1 1 0 1 1 0 1 1 1
Child 2 1 0 1 0 1 1 0 1

Figure 4.6 Single-point crossover
4.6.2 Two-Point Crossover

In two-point crossover, two crossover points are chosen randomly and the contents between these points are exchanged between two mated parents. The advantage of having more crossover points is that the problem space may be searched more thoroughly. In the Figure 4.7, the dotted lines indicate the crossover points.

![Figure 4.7 Two-point crossover](image)

4.6.3 Uniform Crossover

In uniform crossover, each gene in the offspring is created by copying the corresponding gene from one or the other parent chosen according to a random generated binary crossover mask of the same length as the chromosomes. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask the gene is copied from the second parent. A new crossover mask is randomly generated for each pair of parents. Therefore, offspring contain a mixture of genes from each parent. In Figure 4.8, new children are produced using uniform crossover.
For non-binary string representations, such as JOR, OBR, and MBR representations, the following crossovers have been adopted in literature:

**4.6.4 Partial-Mapped Crossover**

Partial-mapped crossover (PMX) was proposed by Goldberg and Lingle (1985). It can be viewed as a variation of two-cut-point crossover by incorporating with a special repairing procedure to resolve possible illegitimacy. The PMX is illustrated in the Figure 4.9. The PMX has the following steps:

1. Select two cut-points along the string at random. The substrings defined by the two cut-points are called the mapping sections.

2. Exchange two substrings between parents to produce offspring.

3. Determine the mapping relationship between two mapping sections.

4. Legalize offspring with the mapping relationship.
4.6.5 Order Crossover

Order crossover (OX) was proposed by Davis (1985). It can be viewed as a kind of variation of PMX that uses a different repairing procedure. Figure 4.10 illustrates the OX. The OX has the following major steps:

1. Select a substring from one parent at random.

2. Produce a child by copying the substrings into the corresponding positions as they are in the parent.

3. Delete all the symbols from the second parent, which are already in the substring. The resultant sequence contains the symbols the child needs.

4. Place the symbols into the empty positions of the child from left to right according to the order of the sequence to produce an offspring.
Falkenauer and Bouffouix (1991) proposed a modified version of the OX which has been called a linear order crossover (LOX). OX tends to transmit the relative positions of genes rather than the absolute ones. In the OX, the chromosome is considered to be circular since the operator is devised for the TSP. The LOX has been developed, where the chromosome is considered linear instead of circular. Figure 4.11 illustrates the LOX. The LOX works as follows:

1. Select sublists from two parents randomly (Sublist1 from P1 and Sublist2 from P2).

2. Remove Sublist2 from parent P1 leaving some 'holes' (marked with h) and then slide the holes from the extremities towards the centre until they reach the cross section. Similarly, remove sublist1 from parent P2 and slide holes to cross section.

3. Insert sublist1 into the holes of parent P2 to form the offspring o1 and insert sublist2 into the holes of parent P1 to form an offspring o2.
4.7 Mutation for JSSP

After crossover operation, the chromosomes are subjected to mutation with a small probability ($P_m$). Mutation prevents the algorithm to be trapped in a local minimum. If crossover is supposed to exploit the current solution to find better ones, mutation is supposed to help for the exploration of the whole search space. Mutation is viewed as a background operator to maintain genetic diversity in the population. It introduces new genetic structures in the population by randomly modifying some of its building blocks. Mutation helps escape from local minima and maintains diversity in the population. There are many different forms of mutations for the different kinds of chromosome representations. For binary representation
such as DGR in section 4.5.1, a simple mutation inverts the value of one or more genes. Mutation of a gene involves flipping its value; changing 0 to 1 and vice-versa. It is relatively easy to make some mutation operators for permutation representations. Several mutation operators have been proposed for permutation representations (JBR, OBR, MBR), such as reciprocal exchange mutation, inversion, insertion, and displacement. Figure 4.12 illustrates these types of the mutations. Reciprocal exchange mutation, Figure 4.12(a), selects two positions at random and then swaps the genes on these positions. Inversion mutation, Figure 4.12(b), selects two positions within a chromosome at random and then inverts the substring between these two positions. Insertion mutation, Figure 4.12(c), selects a gene at random and inserts it in a random position. Displacement mutation, Figure 4.12(d), selects a substring at random and inserts it in a random position.

![Mutations for permutation representation](image-url)

**Figure 4.12** Mutations for permutation representation
4.8 The Proposed Multi-Agent Genetic Algorithm for JSSP

The proposed Multi-Agent Genetic Algorithm (MAGA) is based on the MAS. In MAGA, each agent contains one chromosome that represents one feasible solution (schedule) in the JSSP. The agents have two types of behaviours, the cooperation behaviour and the self-learning behaviour. The cooperation behaviour is incorporated by communications between the agents and their neighbours. The self learning behaviour is achieved by applying a local search procedure on each agent to enhance the quality of its solution. MAGA is genetic algorithm hybridized with local search method.

4.8.1 Agents' Environment

The agents reside in a grid where each cell in the grid contains one agent. By wrapping the borders of the grid around to their opposite borders, the grid forms a torus. This torus is depicted in Figure 4.13. The torus structure provides each agent in it with eight neighbours.

![Figure 4.13: Agents' environment](image-url)
Chapter 4. MAGA for JSSP

In the given environment, three types of communications have been proposed and examined. One agent may communicate with two, four, or eight intermediate neighbours as illustrated in Figure 4.14.

![Figure 4.14 Communication types]

4.8.2 Encoding Method

JSSP is a combination optimization problem, where the feasible solution is the sequence of operations of all jobs. The chromosome is encoded as operation-based representation (OBR), which is one of commonly used encoding methods for JSSP. In OBR, a schedule is described as a sequence of all operations of all jobs, and each operation in the sequence is described by its job-number. Given a JSSP of size $n \times m$, each job in the sequence is repeated $m$ times, where $n$ is number of jobs and $m$ is number of machines. A sequence can be represented as: $S = [s_1, s_2 \ldots s_{n \times m}]$ where $s_k = i$ is an operation from a job $J_i$ represented in the sequence by its job number, $1 \leq k \leq n \times m$, and $1 \leq i \leq n$.

The schedule is obtained by considering the operations in the order of their occurrence in $S$ from left to right and assigning the earliest allowable starting time to that operation. This encoding method has the advantage that no infeasible schedule is represented. Unfortunately, OBR encoding method produces redundancy in the search space of the JSSP where the mapping relation between sequences and schedules in OPR is
many-to-one and will cause the search-space size to expand to \((n \times m)! / (m!)^n\) as stated in (Ge et al., 2008). For example, suppose a JSSP of size 3×3. The sequences \(S_1 = [1, 3, 3, 1, 2, 2, 1, 3, 2]\) and \(S_2 = [1, 3, 3, 1, 2, 2, 1, 2, 3]\) produce equivalent schedules if the last two operations of the sequences \(S_1\) and \(S_2\) are running on different machines.

### 4.8.3 Crossover

The purpose of the crossover operation is to obtain high fitting chromosome from two chosen parent chromosomes. Here Order Crossover (OX) has been used, where two cut points are chosen randomly. The offspring is obtained by copying the middle subsequence from the first parent to the corresponding positions in the offspring. The remaining positions are filled from the second parent in the order from left to right omitting the operations occurring in the middle subsequence.

### 4.8.4 Mutation

Mutation operation deals with one chromosome at a time to get a new offspring. The purpose of the mutation is to preserve the population diversity. Insert and invert mutations are used in this algorithm. In the insert mutation, one gene is selected randomly and inserted to the randomly chosen position. In the invert mutation a substring is chosen randomly and inverted its order.

### 4.8.5 Local Search Procedure

Local search employs the idea that a given solution may be improved by making small change to it. Local search is the important component in the proposed algorithm. It is applied on each offspring generated by genetic
operations to enhance its makespan. The proposed local search is applied on the sequence \( S \) to generate a new improved sequence \( \hat{S} \). The pseudo-code of the local search procedure is presented as follows:

\[
\text{LS\_Procedure}(S, C_{\text{max}})
\]

<table>
<thead>
<tr>
<th>Input: sequence ( S ), makespan ( C_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: sequence ( \hat{S} )</td>
</tr>
</tbody>
</table>

\textbf{for} \( i=1 \) to \( S.\text{length}-1 \) \textbf{do}

\begin{align*}
\text{begin} \\
\text{for} \ j=2 \text{ to } S.\text{length} \text{ do} \\
\text{begin} \\
\text{TE} = \text{Exchange} \ (S, \ s_i, \ s_j) ; \\
M = \text{Makespan} \ (\text{TE}) ; \\
\text{if } (M \leq C_{\text{max}}) \text{ then} \\
\text{begin} \\
S = \text{TE} ; \\
C_{\text{max}} = M ; \\
\text{end if} \\
\text{end if} \\
\text{end for} \\
\text{end for} \\
\hat{S} = S ; \\
\text{return } \hat{S}
\end{align*}

The function \( \text{Exchange}(S, s_i, s_j) \) generates a new sequence by exchanging \( s_i \) and \( s_j \) in the sequence \( S \) and returns the new sequence. The function \( \text{Makespan}(S) \) calculates the makespan for the sequence \( S \). The pseudo-code of the function \( \text{Makespan}() \) is presented as follows:
Makespan(S)

**Input:** the sequence S.

**Output:** the makespan of the sequence S.

// n and m are the number of jobs and machines.
// Jt[i] the time job i has used so far.
// Mt[j] the time machine j has used so far.
// Machine_no() returns the machine number of the given operation.

Initialize the two arrays Jt[n] and Mt[m];
Jt[1..n] = 0 and Mt[1..m] = 0;

for i = 1 to n do
    begin
        j = Machine_no(S[i]);
        St = MAX{Jt[S[i]], Mt[j]};
        Jt[i] = Mt[j] = St + p_{ij};
    end for

makespan = MAX{Jt[i]: i = 1..n}
return makespan

4.8.6 Multi-Agent based Genetic Algorithm

The Multi-Agent based Genetic Algorithm (MAGA) starts with uniformly random initial population. The algorithm evolves according to the following steps using genetic operations and local search method until a fixed number of generations is reached or the optimal solution is found. The flowchart of the MAGA is shown in Figure 4.15.
Chapter 4. MAGA for JSSP

Start

Initialize a population of Agents

G = 0

Evaluate the population

Select agents

Offspring $\leftarrow$ Crossover (Agent, Neighbour)

Mutation (Offspring)

LS_Procedure(Offspring)

Add the offspring to a new population

G = G+1

End

Figure 4.15 Flowchart of the MAGA
The pseudo-code of the MAGA is the followings:

MAGA

**Step1.** Generate initial random population of agents in the grid.

**Step2.** Evaluate the current population.

**Step3.** Apply crossover operation on selected agent using tournament selection and randomly chosen intermediate neighbour agent with probability $P_c$. Replace less fit parents on the grid with the generated offspring.

**Step4.** Apply mutation operation on the offspring with probability $P_m$ to get new offspring, and then replace the agents on the grid with the generated offspring.

**Step5.** Apply inversion operation on the offspring with probability $P_i$ to get a new offspring, and then replace the agent on the grid with the generated offspring.

**Step6.** Apply local search procedure on all new offspring in the population.

**Step7.** Repeat step 2 through 7 until a fixed number of generations is reached or an optimal solution is found.

The MAGA has a hybrid genetic algorithm structure and it uses the elitism concept in the step 3 of the algorithm, where the less fit parent on the grid is replaced with the newly generated offspring and thus the agent which contains the fittest chromosome in the grid is kept intact for the next generation. Thus, the elitism strategy is implemented within the selection process.
4.9 Implementation and Results

The proposed algorithm has been implemented using the agent-based simulation toolkit REPAST from (North et al., 2003). The computing system for the experiments was a computer system which has Intel C2D 1.8 GHz processor, DDR2 2.5 GB RAM, and OS 64-bit Windows Vista.

4.9.1 Parameter Tuning

Empirical experiments have been done to tune the parameters of the MAGA on the well-known FT10 benchmark instance of the JSSP. The crossover rate has been chosen in the range [0.6, 0.9] and the mutation rate in the range [0.0001- 0.001]. The maximum number of generations is 500, the Grid size is 20×20 agents, and communication scheme of the agents with 8-Neighbours has been used. The experiments have been repeated for 50 runs (executions) for each one of the parameter sets. Table 4.2 shows the average minimum makespans (Avg) and the average generation number (AvGn) for some of the experimented parameter sets on the JSSP benchmark instance FT10:

Table 4.2 Parameters tuning on FT10: \( P_c \) and \( P_m \)

<table>
<thead>
<tr>
<th>Parameter sets</th>
<th>Avg</th>
<th>AvGn</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_c = 0.6, P_m = 0.0001 )</td>
<td>930</td>
<td>17.33</td>
</tr>
<tr>
<td>( P_c = 0.7, P_m = 0.0001 )</td>
<td>930</td>
<td>17.66</td>
</tr>
<tr>
<td>( P_c = 0.8, P_m = 0.0001 )</td>
<td>930</td>
<td>22.33</td>
</tr>
<tr>
<td>( P_c = 0.9, P_m = 0.0001 )</td>
<td>930</td>
<td>19.8</td>
</tr>
<tr>
<td>( P_c = 0.6, P_m = 0.0005 )</td>
<td>930</td>
<td>17.66</td>
</tr>
<tr>
<td>( P_c = 0.6, P_m = 0.001 )</td>
<td>930</td>
<td>19.22</td>
</tr>
<tr>
<td>( P_c = 0.6, P_m = 0.005 )</td>
<td>930</td>
<td>43.03</td>
</tr>
</tbody>
</table>
In Table 4.2, the average minimum makespan is 930 for all the parameter sets. This is due to the efficiency of the MAGA in solving the FT10 instance where the MAGA algorithm reaches the best known solution for FT10 (i.e. 930) in every run. However, the average generation number (AvGn) differs for each parameter set in the Table 4.2. The stochastic behavior of the MAGA makes the algorithm reach the best known solution with different generation number in each run. The parameter set $P_c = 0.6$, $P_m = 0.0001$ is chosen in the first row of the Table 4.2 to be used henceforth, since the MAGA algorithm achieves the least average generation number with this set.

Other important parameter in the MAGA model is the communication scheme between the agents. Three communication schemes have been proposed and implemented: MAGA2 uses 2-Neighbours communication, MAGA4 uses 4-Neighbours communication, and MAGA8 uses 8-Neighbours communication. Figure 4.16 shows the comparison between the results obtained from the three communication schemes. The MAGA is executed for 50 times on four benchmark instances of the JSSP from OR library (FT10, FT20, LA16, LA26). All these instances are solved efficiently, where the best known solution is reached in each run of the MAGA model. It has been noticed from Figure 4.16 that the convergence to the best makespans for all the benchmark instances is faster for MAGA8 than for MAGA2 or MAGA4. The reason is that the probability of the propagation of the better solution in the grid for 8-Neighbours communication scheme is faster than other proposed communication scheme (2-Neighbours and 4-Neighbours). Henceforth, the MAGA8 communication scheme is considered to be used in the MAGA.
4.9.2 Experimental Results

The proposed MAGA has been tested on eleven benchmark instances that have been taken from OR library. The experiments have been repeated for 50 runs for each benchmark instance, with number of generations up to 500, Grid size as 20×20 agents, Crossover rate as 0.6, and Mutation rate as 0.0001. The communication scheme MAGA8 has been used. The parameters settings for MAGA are given in the Table 4.3. It has been noticed from the experiments that there is no further improvement in the minimum makespan after 500 generations.

Table 4.3 Parameters settings for MAGA

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum generations</td>
<td>500</td>
</tr>
<tr>
<td>Population size</td>
<td>400</td>
</tr>
<tr>
<td>Crossover rate ($P_c$)</td>
<td>0.6</td>
</tr>
<tr>
<td>Mutation rate ($P_m$)</td>
<td>0.0001</td>
</tr>
</tbody>
</table>
Chapter 4. MAGA for JSSP

The results obtained from MAGA have been compared with the results from MAEA-JSPs by (Zhong et al., 2005) in Table 4.4 in terms of Best, Avg, and AvGen. In Table 4.4, Bk is the best known makespan from the literature, Best is the best minimum makespan obtained from all the 50 runs of the algorithm, Avg is the average minimum makespan obtained from all the 50 runs of the algorithm, AvGn is the average generation number to get the best known makespan from all the 50 runs of the algorithm, and the gap G that has been calculated according to the following equation:

\[ G = \frac{(Avg - Bk)}{Bk} \times 100\% \]  

(4.1)

Table 4.4 The experimental results of MAGA and MAEA-JSPs

| Benchmark | Bk | MAEA-JSPs | MAGA | | | | |
|-----------|----|-----------|------|---|---|---|---|---|---|---|---|
|           |    | Best | Avg       | G   | Best | Avg       | G   | AvGn | |
| FT06      | 55 | 55   | 55        | 0   | 55   | 55        | 0   | 1   |
| FT10      | 930| 930  | 944.45    | 1.55| 930  | 930       | 0   | 17.33|
| FT20      | 1165|1165|1178.89   | 1.19|1165 |1165      | 0   | 42.2 |
| LA01      | 666| 666  | 666       | 0   | 666  | 666       | 0   | 1   |
| LA06      | 926| 926  | 926       | 0   | 926  | 926       | 0   | 1   |
| LA11      | 1222|1222|1222      | 0   | 1222 |1222      | 0   | 1   |
| LA16      | 945| 945  | 945       | 0   | 945  | 945       | 0   | 11.1 |
| LA21      | 1046|1046|1068.11   | 2.11|1046 |1047.8    | 0.17|59   |
| LA26      | 1218|1218|1219.15   | 0.09|1218 |1218      | 0   | 7.33 |
| LA31      | 1784|1784|1784     | 0   | 1784 |1784      | 0   | 1.86 |
| LA36      | 1268|1274|1295.49   | 2.17|1268 |1272.56   | 0.36|220.3|

Following all the observations from Table 4.4:

- For the instance FT06, both of the MAEA-JSPs and the MAGA attains the best known makespan (i.e. 55) in all the runs as the
average minimum makespan is 55 and the gap is 0. The average generation number attained by the MAGA is 1.

- For the instance **FT10**, the MAGA attains the best known makespan (i.e. 930) in all the runs as the average minimum makespan is 930 and the gap is 0, while the MAEA-JSPs is not as the average minimum makespan is 944.45 and the gap is 1.55. The average generation number attained by the MAGA is 17.33.

- For the instance **FT20**, the MAGA attains the best known makespan (i.e. 1165) in all the runs as the average minimum makespan is 1165 and the gap is 0, while the MAEA-JSPs is not as the average minimum makespan is 1178.89 and the gap is 1.19. The average generation number attained by the MAGA is 42.2.

- For the instance **LA01**, both of the MAEA-JSPs and the MAGA attains the best known makespan (i.e. 666) in all the runs. The gap is 0 for the MAEA-JSPs and the MAGA. The average generation number attained by the MAGA is 1.

- For the instance **LA06**, both of the MAEA-JSPs and the MAGA attains the best known makespan (i.e. 926) in all the runs as the average minimum makespan is 926 and the gap is 0. The average generation number attained by the MAGA is 1.

- For the instance **LA11**, both of the MAEA-JSPs and the MAGA attains the best known makespan (i.e. 1222) in all the runs as the average minimum makespan is 1222 and the gap is 0. The average generation number attained by the MAGA is 1.

- For the instance **LA16**, both of the MAEA-JSPs and the MAGA attains the best known makespan (i.e. 945) in all the runs as the
average minimum makespan is 945 and the gap is. The average generation number attained by the MAGA is 11.1.

- For the instance LA21, both of the MAEA-JSPs and the MAGA attain the best known makespan (i.e. 1046) in some of the runs. The average minimum makespan is 1068.11 and the gap is 2.11 in the MAEA-JSPs, while the average minimum makespan is 1047.8 and the gap is 0.17 in the MAGA. This yields that the MAGA is better than the MAEA-JSPs in approaching to the best known makespan. The average generation number attained by the MAGA is 59.

- For the instance LA26, the MAGA attains the best known makespan (i.e. 1218) in all the runs as the average minimum makespan is 1218 and the gap is 0 while the MAEA-JSPs attain the best known makespan in some of the runs as the average minimum makespan is 1219.15 and the gap is 0.09. Thus, the MAGA is better than the MAEA-JSPs for solving this instance. The average generation number attained by the MAGA is 7.33.

- For the instance LA31, both of the MAEA-JSPs and the MAGA attain the best known makespan (i.e. 1784) in all the runs as the average minimum makespan is 1784 and the gap is 0 for the MAEA-JSPs and the MAGA. The average generation number attained by the MAGA is 1.86.

- For the instance LA36, the MAGA attains the best known makespan (i.e. 1268) in some of the runs while the MAEA-JSPs does not. The average minimum makespan is 1295.49 and the gap is 2.17 in the MAEA-JSPs while the average minimum makespan is 1272.56 and the gap is 0.36 in the MAGA. Thus, the MAGA is
better than the MAEA-JSPs in solving this JSSP benchmark instance. The average generation number attained by the MAGA is 220.3.

In summary, The MAGA attains the best known makespan for the eleven benchmark instances while the MAEA-JSPs algorithm fails to attain the best known makespan for one JSSP benchmark instance (i.e. LA36) but attains the best known makespan for other ten benchmark instances. It is observed that the proposed model MAGA yields better performance for the FT10, FT20, LA21, LA26, and LA36. Because, the average minimum makespan and gap values obtained in these benchmarks by MAGA were lower than the corresponding values obtained by MAEA-JSPs.

Figure 4.17 shows the implementation of MAGA algorithm in the REPAST Toolkit for the instance FT20, where the graph shows that the minimum makespan achieved at 52\textsuperscript{nd} iteration is 1165 which is equal to the best known makespan for this instance. Other components of the Figure 4.17 are the agent grid and the output window. The agents are graphically represented as coloured squares in a grid. The output window of REPAST shows the current best makespan, average makespan, and the current generation number during the run of the MAGA model.

4.10 Conclusion

The features of the two models MAS and GA have been integrated in the proposed MAGA model. Further, the self-learning behaviour of agents has been implemented by the local search technique. The MAGA has been tested on a set of 11 standard JSSP instances taken from the OR-Library. The results obtained from MAGA have been compared with MAEA-JSPs.
algorithm from a published paper. The results indicate that MAGA is more successful than MAEA-JSPs in terms of attaining the best known makespan for all the eleven JSSP benchmark instances and in terms of the average makespan and gap for the experimented JSSP benchmark instances FT20, LA21, LA26, LA36. The MAGA has been published as conference paper in (Balid and Minz, 2008a) and in (Balid and Minz, 2008b).

Figure 4.17 MAGA Model in the REPAST toolkit